Detecting Flaky Tests by Controlling Nondeterministic API Behavior

HENGCHEN YUAN, The University of Texas at Austin, USA
JIEFANG LIN, The University of Texas at Austin, USA
AUGUST SHI, The University of Texas at Austin, USA

Regression testing is an essential part of software development to ensure high-quality software; but the presence of flaky tests makes the testing outcomes unreliable. It is essential to proactively detect flaky tests, so developers are aware of them early on and can react appropriately in case they fail. While there has been prior work in detecting flaky tests, they either are developed to focus on specific flaky test types or can be imprecise in their analyses, such as by relying on AI models for prediction.

We present ChaosAPI, a framework to support detecting a variety of different types of flaky tests. Our insight is that the most effective approach to detecting flaky tests is to target the nondeterministic components that lead tests to have the flaky behavior in the first place. In particular, we target specific APIs within the Java Standard Library that all Java code relies on and that are known to exhibit nondeterministic behavior, such as those related to system time, concurrency, and environmental factors. During test execution, ChaosAPI modifies the behavior of these API calls, perturbing inputs and return values of these APIs in a systematic manner while still remaining compliant with the API specification. We can detect flaky tests by observing whether tests that previously passed would now fail when run through ChaosAPI.

Our evaluation on a prior dataset of known flaky tests, as well as running on test suites of other popular open-source projects, demonstrates that ChaosAPI not only detects more flaky tests than simple rerunning across a wide range of projects but also detects them more efficiently, making ChaosAPI a practical addition to the toolbox of flaky test detection techniques.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

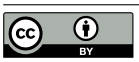
Additional Key Words and Phrases: flaky tests, nondeterminism, API perturbation

ACM Reference Format:
Hengchen Yuan, Jiefang Lin, and August Shi. 2026. Detecting Flaky Tests by Controlling Nondeterministic API Behavior. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 157 (April 2026), 27 pages. https://doi.org/10.1145/3798265

1 Introduction

Regression testing is an important part of the software development process, but it suffers from the presence of flaky tests. A *flaky test* is a test that can nondeterministically pass or fail when run on the same code, without any changes [10, 23]. When a flaky test fails when run after changes as part of regression testing, its failure can mislead the developer into believing their changes introduced bugs [6]. Flaky tests are a serious problem for developers in both open-source and industry [4, 13–16, 41]. Prior work has proposed many ways to proactively detect flaky tests, so developers are aware of their existence ahead of time, by focusing on specific types of flaky

Authors' Contact Information: Hengchen Yuan, The University of Texas at Austin, Austin, TX, USA, hcyuan@utexas.edu; Jiefang Lin, The University of Texas at Austin, Austin, TX, USA, jiefang@utexas.edu; August Shi, The University of Texas at Austin, Austin, TX, USA, august@utexas.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/4-ART157
https://doi.org/10.1145/3798265

tests [9, 17, 20, 34] or relying on machine learning models to predict which tests are flaky without rerunning the tests [5, 11].

Tests can be flaky for many different reasons, such as due to asynchronous waits, concurrency, test-order dependencies, or random numbers [10, 23]. At its core, there is some nondeterministic component during execution that is uncontrolled by the test, whose output or behavior eventually results in the test having differing outcomes even when run on the same code. For example, a random number generator, when the initial seed is uncontrolled, can output just about any number, but certain numbers or sequences of numbers may lead to the test behaving differently, resulting in an unexpected failure. Indeed, there may be many library-level APIs that have uncontrolled, nondeterministic outputs that lead to flaky behavior, e.g., Shi et al. previously studied how certain Java Standard Library APIs that involve iterating over data structures like `Maps` do not guarantee the order in which to iterate over the elements, but some tests make assumptions that the iteration order is deterministically a specific ordering [34]. Such wrong assumptions lead the tests to fail when the iteration order is changed.

Our insight is that we can proactively detect flaky tests by directly controlling the sources of nondeterminism, specifically the Java Standard Library-level APIs. While controlling nondeterminism has been explored in prior work, existing techniques typically focus on specific and well-studied sources, such as test-order dependency [17], asynchronous behavior [33], or data-structure iteration [34]. However, there has been no systematic study of how uncontrolled API-level behavioral nondeterminism impacts flaky test failures in practice, including its prevalence, manifestations, and interactions with existing flakiness categories. Our work targets this broader and complementary class of API-level behavioral nondeterminism.

We implement ChaosAPI, a framework for controlling the behavior of nondeterministic APIs within the Java Standard Library. ChaosAPI instruments the call sites to target APIs during test execution, changing their behavior. Instead of directly changing Java Standard Library implementations of those APIs, ChaosAPI rewrites the API calls to the corresponding APIs defined within ChaosAPI, which are simply wrappers around the original APIs but instead allow changing of the API's inputs and outputs. We define several *strategies* that perturb either the output or input of an API, taking care that our perturbation respects the API's specification. Our goal is to explore different acceptable behaviors by the APIs and observe how the rest of the execution and finally the test outcomes can be affected by those changed behaviors. In particular, we want to allow for configuring the API perturbation to explore extreme boundary conditions, representing more unexpected behavior which should more easily induce flaky test failures. Tests that have different outcomes under different perturbed API behavior are then detected as flaky tests. A developer can define several different strategies that target specific APIs and perturb their inputs/outputs. For our own implementation and evaluation, we develop 11 strategies that target 17 different Java Standard Library APIs.

We evaluate ChaosAPI on a dataset of known flaky tests from prior work [5]. This dataset consists of flaky tests detected through rerunning tests from open-source projects 10,000 times, saving any failure logs corresponding to flaky test failures. We successfully built and ran the tests for 11 projects from this dataset, which contains 87 known flaky tests. By running these projects' tests through ChaosAPI, we detect 61 flaky tests, which we confirmed to correspond to known flaky tests from the prior dataset. Beyond these known flaky tests, ChaosAPI detects 534 flaky tests in total across all tests from these projects. Our inspection of all these failures showed that 507 of those tests are truly flaky. Beyond confirming the detection results, we also perform an extensive manual inspection and validation of observed failures to understand how API-level nondeterminism manifests in practice and how it relates to other well-known flakiness categories.

In inspecting the 27 failing tests we deemed to be false positives, we find that the main reasons for them failing yet not being flaky arise from two sources: (i) inconsistency in expected behavior between related APIs, where some we perturb through our strategies but the others we do not (e.g., third-party wrappers or uncommon APIs we do not currently handle), and (ii) the perturbations by some of our strategies end up making drastic changes to behavior beyond what could be considered reasonable, such as changing time APIs like `System.currentTimeMillis()` to return values that are much too large and different from the actual time. Given that our strategies are configurable, a user with more in-depth domain knowledge of the project can provide better configuration values that are less likely to result in false positives.

We further run ChaosAPI on a larger set of 24 popular open-source projects to see whether it can detect new, unknown flaky tests in projects outside of the existing benchmark dataset. ChaosAPI detects 482 new potential flaky tests, where we confirmed 300 are real in a sampled subset while simply rerunning tests normally can only detect 12 within the same amount of time.

Our paper makes the following contributions:

- We present the first systematic study of the impact of API-level behavioral nondeterminism on flaky test failures, focusing on nondeterministic behaviors exposed through commonly used library APIs that are not explicitly controlled in existing test suites.
- We propose a specification-respecting perturbation methodology and implement ChaosAPI, a lightweight and extensible framework that exposes API-level nondeterminism by perturbing the outputs or input of Java Standard Library APIs, without requiring test rewriting or explicit mocks. ChaosAPI supports 11 perturbation strategies across 17 commonly used Java Standard Library APIs.
- Through extensive evaluation, including manual inspection and validation of detected failures, we characterize the prevalence, distribution, failure manifestations, and relationships of API-level behavioral nondeterminism with existing flakiness categories.
- We evaluate ChaosAPI on a dataset of previously known flaky tests as well as on a larger set of popular open-source projects with potentially unknown flaky tests. ChaosAPI showed a precision of 94.9% and recall of 70.1% on this dataset of known flaky tests. It also detects 746 manually confirmed previously unknown flaky tests.

2 Example

Fig. 1 shows an example flaky test from project DiUS/java-faker [8]. This test invokes `faker.sip().rtpPort()` (line 4), which delegates to `Sip.rtpPort()` (line 14). Within the method `rtpPort()`, an index `idx` is obtained via `faker.random().nextInt(0, portPool.size())` (line 16) and then used in `portPool.get(idx)` (line 17). The helper `RandomService.nextInt(min, max)` (line 30) implements an *inclusive* upper bound, i.e., `random.nextInt((max - min) + 1) + min`, so it may return any value in $[min, max]$. With $min = 0$ and $max =$ `portPool.size()`, the returned value can equal `portPool.size()`, i.e., one past the list's largest valid index (`portPool.size()` $- 1$). When this boundary case occurs, `portPool.get(idx)` (line 17) throws an `IndexOutOfBoundsException`. We can compute the probability of the random value returned being out of bounds and triggering the test failure to be $1/($`portPool.size()` $+ 1)$, which is less than 0.02%. Note that this behavior can happen without any changes to the code, but only sometimes, so the test by definition is flaky.

While the failure is relatively rare, we can induce the failure to occur with a much higher chance through perturbing the outputs of the relevant methods. Fig. 2 illustrates this process of intercepting the method call to return a different value. More specifically, we change the call to `Random.nextInt(int)` to instead use a custom API of our own, `ChaosAPI.nextInt(int)`. Our custom API consistently returns a different number than what the built-in random number generator would

```java
/* From java-faker#SipTest (trimmed for clarity) */
@Test
public void rtpPort_returnPositiveEvenInt() {
    int sut = faker.sip().rtpPort(); // <-- Triggering IndexOutOfBoundsException with
     ↪ extremely low probability.
    assertThat(sut, greaterThanOrEqualTo(2));
    assertThat(sut % 2, is(0));
}

/* Code Under the Test */
public class Sip {
    private final Faker faker;
    private final java.util.List<Integer> portPool; // [40000, 40002, ...., 50000]

    public int rtpPort() {
        // If helper is inclusive, nextInt(0, size) may return 'size'
        int idx = faker.random().nextInt(0, portPool.size()); // <-- hazard: [0, size]
        return portPool.get(idx);  // <-- get(size) => IndexOutOfBoundsException
    }
}

public class Faker {
    private final RandomService randomService;
    public RandomService random() { return this.randomService; }
    public Sip sip() { return new Sip(this); }  // factory; details omitted
}

public class RandomService {
    private final java.util.Random random;       // JDK Random
    // Inclusive helper: returns in [min, max]
    public Integer nextInt(int min, int max) {
        return random.nextInt((max - min) + 1) + min;  // <-- includes 'max'
    }
}
```

Fig. 1. Trimmed real-world code snippet from java-faker (unrelated parts omitted)

return. Note that this behavior can still be considered acceptable, since there are no guarantees as to the types of numbers a random number generator must return or the distribution of possible returned numbers. If we configure the execution to have `ChaosAPI.nextInt(int)` always return the largest acceptable value (namely `max-min` in this example), we would observe the test constantly failing under those conditions.

Our goal is to develop a framework that can perturb APIs in this manner, allowing us to control the behavior of similar APIs that are typically outside the control of the test code, to explore how their different execution behaviors can affect the test outcomes and expose flaky tests.

3 ChaosAPI Framework

We present ChaosAPI, a framework for systematically detecting flaky tests. The core of ChaosAPI comes from a set of *strategies* that we define for perturbing the behavior of target APIs from the Java Standard Library that exhibit some form of nondeterminism or behavior uncontrolled by the test code, e.g., random number generation, system time, concurrency constructs, etc. More precisely, a strategy targets a specific set of one or more APIs with nondeterministic behavior, while also

```java
public int rtpPort() {
    int idx = faker.random().nextInt(0,
     ↪  portPool.size());
    return portPool.get(idx);
}
// RandomService.nextInt(min, max)
public Integer nextInt(int min, int max
     ↪ ) {
    // Calls JDK's Random
    return random.nextInt((max-min)+1)+
     ↪ min;
}
```

Standard Runtime Behavior (Flaky)

Execution Flow: Our tool intercepts API call of `Random.nextInt(bound)`, perturb its return value to the allowed maximum `bound-1`, making the `IndexOutOfBoundsException` occur deterministically.

```java
public int rtpPort() {
    int idx = faker.random().nextInt(0,
     ↪  portPool.size());
    return portPool.get(idx);
}
// RandomService.nextInt(min, max)
public Integer nextInt(int min, int max
     ↪ ) {
    -TOOL INTERCEPTION POINT-
    return ChaosAPI.nextInt(max-min+1)+
     ↪ min;
}

// ChaosAPI.nextInt(bound):
public int nextInt(int bound){
    return bound - 1;
}
```

Instrumented Runtime Behavior (Deterministic)

Fig. 2. Side-by-side comparison of the **runtime behavior** of the original code. The source code remains unchanged, but our tool intercepts the call to `random.nextInt` to make the test failure deterministic.

sharing either the same return type or input parameter. A perturbation specifically changes either the return value of the target APIs or an input value, allowing any usage of that API to exhibit different behavior but still remain compliant to the API specifications.

ChaosAPI implements a strategy within its framework by instrumenting calls to the target APIs. For each target API, ChaosAPI constructs a wrapper API that can call the corresponding target API but perturbs the return value or input value. For return value perturbation, the wrapper API calls the target API with the same input but changes the return value in a controlled manner. For input value perturbation, the wrapper API changes the input value in a controlled manner and calls the target API with the new input and outputs what the target API returns. ChaosAPI then leverages dynamic bytecode instrumentation to replace calls to target APIs with their corresponding wrapper APIs. ChaosAPI executes tests in this environment where they are calling these wrapper APIs instead of the original target APIs. Tests that fail are likely flaky tests making unwarranted assumptions about the behavior of these target, nondeterministic APIs.

Fig. 3 shows a basic workflow of the ChaosAPI framework. Developers first choose the perturbation strategies to enable and their corresponding configurable parameters, which control how the return values and input values are perturbed. We provide an initial set of perturbation strategies (Section 4), but a developer can define new custom strategies to include into ChaosAPI. ChaosAPI then applies each strategy to the project code, using a Java instrumentation library to dynamically modify and replace all target API calls with the corresponding wrapper APIs. As classes are loaded by the JVM during test execution, ChaosAPI locates the call sites of all target Java Standard Library APIs by all strategies, replacing each call with the corresponding wrapper API. Note that ChaosAPI can also perturb constructors as target APIs, in which case ChaosAPI employs a factory pattern where bytecode instructions to construct new objects are replaced with calls to a static factory method that creates the object instance, allowing ChaosAPI to further customize and perturb the returned object. For most API calls, ChaosAPI simply replaces calls to the target APIs with wrapper

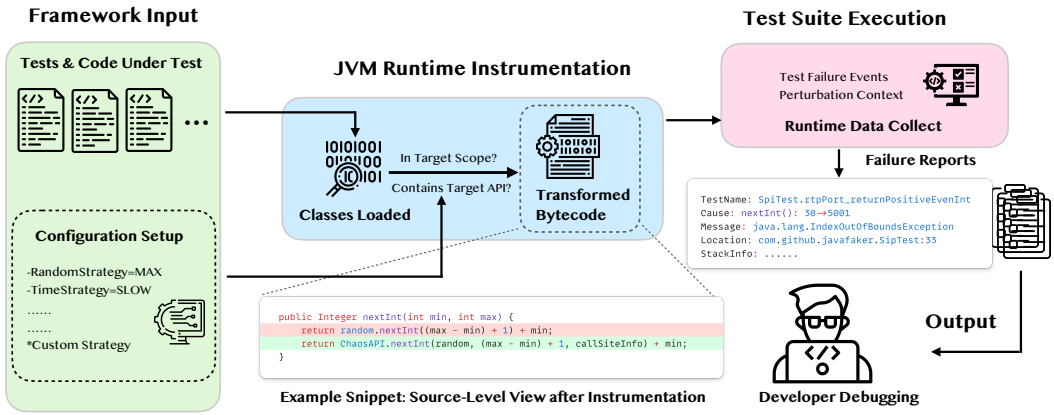


Fig. 3. An overview workflow of ChaosAPI

APIs, allowing it to focus on replacing only those APIs used within the project code itself instead of calls to those APIs in outside third-party libraries or even within the Java Standard Library itself. We want to only perturb calls to these APIs within project code to avoid unintended side-effects from external library usages of these APIs. For some APIs representing behavior that is more global to the entire execution environment, ChaosAPI can modify the callee itself, perturbing all calls to that API, regardless of coming from project code or even the Java Standard Library itself.

ChaosAPI applies all given strategies together on the same code. Note that if two strategies have overlapping target APIs, ChaosAPI would not be able to apply both at the same time. As tests execute once the code has been dynamically instrumented, any test failures at the end represent potential flaky tests that are finally reported to the developer for inspection. ChaosAPI reports the failing test along with the failure message (including stack trace) and locations of all target APIs that were perturbed by the given strategies.

4 ChaosAPI Strategies

We predefine 11 perturbation strategies based on common APIs from the Java Standard Library that have known nondeterministic or uncontrolled behavior. We categorize and present the strategies into 7 groups where the strategies in the same group have the same target APIs. We further present these groups based on whether their strategies perturb the target API return values or input values.

4.1 Return-Value Perturbation Strategies

This category of strategies intercepts calls to target APIs and perturbs their return values to some other, user-configurable values. By changing the default values to some other values that a user might judge to be an edge case for those APIs, these strategies can detect which tests may be relying on common-case behavior of these APIs.

4.1.1 Random Number Generation. Random number generation (RNG) is a common source of non-determinism, as it literally returns any random sequence of numbers that may affect execution [9, 23]. Strategies involving RNG target specific APIs within the `Random` class or the `ThreadLocalRandom` class, both classes that are commonly used to generate random numbers. Specifically, we target common random number generation APIs, including `Random.nextInt()`, `Random.nextInt(int)`, `Random.nextLong()`, `Random.nextDouble()`, `Math.random()`, as well as `ThreadLocalRandom.nextInt()`,

`ThreadLocalRandom.nextInt(int)`, `ThreadLocalRandom.nextInt(int, int)`, `ThreadLocalRandom.nextLong()`, and `ThreadLocalRandom.nextDouble()`.

The RNG specification we comply with is that these APIs return values that lie within a valid range (e.g., `nextInt(n)` $\in [0, n)$), but they do not require any particular distribution of numbers returned. Furthermore, to respect developer-intended determinism, the return value of APIs coming from `Random` instances explicitly initialized with a fixed seed should not be perturbed.

- **Deterministic Value Strategy.** We define this strategy to address any potential assumptions about some distribution of values obtained from RNG, since RNG is not required to ever change values between calls (it could theoretically always return the same value). The Deterministic Value Strategy changes the outputs to the target RNG APIs to always output a user-configured value V. To remain specification-compliant, we ensure that the returned value is always within the specified bounds, e.g., a bounded API call like `Random.nextInt(n)` expects that the value to be less than `n`, so we would always return `n-1` if the user configures the strategy to return a larger value.

4.1.2 Time. APIs that return the system time, namely `System.currentTimeMillis()` and `System.nanoTime()`, would not always return some consistent values, by design. Tests may rely on the outcomes of these method calls, such as using them to calculate the time it takes to perform some computations, using them to compute timeouts.

The specification we comply with is that these APIs can return any number, representing some number of milliseconds or nanoseconds since an initial baseline point like epoch 1970, but subsequent calls to these APIs should never result in a number smaller than what was returned before. Essentially, we must ensure that time moves forward and not backwards, though it may not always move forward by the true amount, especially as system clocks offer limited precision and no guarantee of accuracy [25]. Such perturbations allow us to "trick" the tests into believing certain amount of time has passed, exposing potential timing-related assumptions that tests make. Changing the time in certain ways can be similar to adding delays to slow down parts of execution, in case tests expect those parts of execution need to take a specific amount of time [18, 32], without actually needing to add delays that would slow down execution. Note when we use a Time Strategy, ChaosAPI ignores calls originating from within other classes inside Java Standard Library (such as `Date` constructor) to ensure the independence between each strategy (as we have another strategy that targets `Date` APIs).

- **Fixed Increment Time Strategy.** This strategy detaches time from the system clock, advancing it by a fixed, configurable delta (δ_{time}) upon each API call. It is designed to expose tests that implicitly rely on non-uniform time progression or expected running time range. The new time, t_{new}, is calculated based on the previous time t_{prev}:

$$t_{\text{new}} = t_{\text{prev}} + \delta_{\text{time}}$$

  It remains specification-compliant as the new time is always bigger than the previous.
- **Time Rounding Strategy.** This strategy simulates a low-resolution clock by rounding times to a configurable granularity (G), exposing flaky tests that assume fine precision for time or assume elapsed time must be greater than 0. To calculate the new time, t_{new}, the actual system time (t_{real}) is rounded to the nearest multiple of G.

$$t_{\text{new}} = \text{round}\left(\frac{t_{\text{real}}}{G}\right) \times G$$

  It remains specification-compliant as the new time is always a rounded-up version of the previous time, and we take the maximum against the previously returned value t_{prev}.

- **Time Scaling Strategy (Fast).** This strategy scales the passage of time by a configurable factor (S_{fast}), making the clock appear to run faster ($S_{\text{fast}} > 1$). It exposes tests with fragile timeout assertions or dependencies on execution speed and machine configurations. The t_{new} time is calculated by scaling the real elapsed time since the test began at t_{start}:

$$t_{\text{new}} = t_{\text{start}} + \big((t_{\text{real}} - t_{\text{start}}) \times S_{\text{fast}}\big), \quad S_{\text{fast}} \geq 1$$

  It remains specification-compliant as the new time is always bigger than the previous, just by a different scale.
- **Time Scaling Strategy (Slow).** Similar to the previous one, this strategy slows the perceived clock to emulate a poorly provisioned or lagging environment.

$$t_{\text{new}} = t_{\text{start}} + \frac{t_{\text{real}} - t_{\text{start}}}{S_{\text{slow}}}, \quad S_{\text{slow}} \geq 1$$

  It remains specification-compliant as the new time is always bigger than the previous, though not as far forward as the current time.

4.1.3 Date. Tests may run on any calendar date and at any point in time. If a test implicitly assumes the current date, we perturb the perceived "now" to control this source of nondeterminism and observe whether the test exhibits different behavior. We specifically target the constructor `Date()`, so any call to construct a new `Date` instance can be perturbed to return a different `Date`.

The specification we comply with is that these APIs can return a valid `Date` object, representing any point of time. There are very few restrictions needed to be placed on how strategies perturb the return value.

- **Fixed Moment Strategy.** This strategy changes the return value of `new Date()` that gets the current date to always return a specific, configurable date D_{target}.

4.1.4 Locale. Similar to the current date, the default locale may vary across execution environments (e.g., different regional/language settings). Locales influence text formatting, number parsing, and case conversions, so tests can exhibit locale-dependent behavior. We specifically target the `Locale.getDefault()` API that returns the current, default locale, usually obtained from the underlying system.

The specification we comply with is that the API returns a valid `Locale` instance, representing a valid locale in the world. There are very few restrictions needed to be placed on how strategies perturb the return value.

- **Locale Change Strategy.** This strategy simply changes the output of `Locale.getDefault()` to other locale L_{target}, configured by the user. Note that this is the only strategy that targets the *callee* to enable all internal APIs within the Java Standard Library to also use the different locale.

4.2 Input Perturbations

This category of strategies changes the input parameters to target API calls. Many of these APIs do not have return values, but by perturbing their inputs, we are able to change their execution behavior, which can in turn affect test executions and induce flaky failures. While we are changing inputs, we ensure that the changes we make still help preserve the specifications of each API.

4.2.1 Thread Sleep. Developers might use `Thread.sleep(long)` to set some delay, waiting for some other parallel computation to complete. Such behavior is common for the Async Wait category of flaky tests [23, 33]. Since external factors like system load or thread scheduling can affect computation time, tests fail if they expect certain computations to take a set amount of time. Indeed, prior studies have shown that the failure rate of some flaky tests is correlated with the underlying

machine configuration [36, 40], as different hardware naturally yields different execution speeds and scheduling behaviors. By perturbing the sleep duration, our strategy proactively simulates these environmental variations to expose such fragile timing assumptions.

The specification we comply with is that the input argument is non-negative and that the `Thread.sleep()` call suspends the current thread for at least the specified duration. It is worth noting that even a call such as `Thread.sleep(0)` is not equivalent to skipping the call entirely. The method still performs interruption checks and may trigger scheduling effects. Thus, our perturbations alter only the length of the suspension request, without changing whether the call itself is executed.

- **Sleep Increase Strategy.** Since `Thread.sleep(long)` takes an input value representing the amount of time to sleep, we perturb that wait time by *increasing* the input with a configurable, non-negative parameter I_{sleep}. Formally,

$$T_{\text{perturbed}} = T_{\text{original}} + I_{\text{sleep}}, \qquad I_{\text{sleep}} \geq 0.$$

  This change simulates what happens when the machine runs slower, needing more time before getting back to wake up the sleeping thread. This strategy is specification-compliant because it still aims to wait at least as long as the original given parameter.
- **Sleep Decrease Strategy.** We also allow *decreasing* the input with a configurable, non-negative parameter D_{sleep} to effectively shorten the sleep, while enforcing a non-negative lower bound:

$$T_{\text{perturbed}} = \max(T_{\text{original}} - D_{\text{sleep}}, 0), \qquad D_{\text{sleep}} \geq 0.$$

  While the specification of `Thread.sleep(long)` intends the actual suspension to be at least the requested duration, we additionally consider a controlled decrease mode that shortens the requested delay to emulate scenarios where the waiting thread resumes earlier while the awaited computation lags. This testing-oriented perturbation helps expose Async Wait-style flakiness without asserting a change to the API's contract; prior work has used shorter delays to expose such failures [23]. This strategy is the only one that is technically not specification-compliant.

4.2.2 Concurrency Primitives with Timeouts. There are several APIs related to concurrency and multiple threads. More specifically, we target the APIs `Object.wait(long)`, `Thread.join(long)`, `CountDownLatch.await(long, TimeUnit)`, `Future.get(long, TimeUnit)`, `Semaphore.tryAcquire(long, TimeUnit)`, `BlockingQueue.poll(long, TimeUnit)` and `Process.waitFor(long, TimeUnit)`. The main commonality among these APIs is that they take a timeout parameter specifying how long the call should wait *at most* for a certain operation or condition to complete. Varying this timeout can simulate environments where concurrent computations take different and uncontrolled amounts of time.

The specification of these timeout-based methods that we comply with is that the call should wait no longer than the given value, possibly returning earlier if the awaited condition completes sooner. Any perturbation should make sure the calls still happen, the timeout argument remains non-negative, yet be at most the original set amount.

- **Concurrency Timeout Shrinking Strategy.** This strategy changes the developer-specified timeout values for all the APIs, T_{original}, to scale down based on the user-configured scaling factor S_{con}, where $0 < S_{\text{con}} < 1$:

$$T_{\text{perturbed}} = T_{\text{original}} \times S_{\text{con}}$$

  It remains specification-compliant because the new timeout value is always less than the original due to scaling down.

4.2.3 Network. Interacting with the network is another common source of flakiness for tests [10, 23]. We specifically target the core connection API in the Java Standard Library, namely the `Socket.connect` API that represents obtaining a blocking TCP connection. Other APIs for blocking TCP connections would ultimately still need to go through this fundamental Java Standard Library API to do so. In particular, we are concerned with the second parameter, the timeout value, which is the maximum amount of time the API waits as it tries to form the socket connection.

The specification we comply with is that the `Socket.connect` call waits at most the requested timeout, where it may return earlier on success or failure. The call must still happen, the timeout should be non-negative, and the timeout must be at most the original set amount.

- **Network Delay Strategy.** Similar to the concurrency timeout perturbation, we modify the timeout input to explore timing-related nondeterminism; however, unlike those APIs, network operations inherently incur external I/O latency, so we introduce a bounded pre-delay to model a slow network path while keeping the observable blocking budget comparable to the original configuration. More specifically, we scale down the developer-specified timeout T_{original} by a user-configured factor S_{net} with $0 < S_{\text{net}} < 1$:

  $$T_{\text{perturbed}} = T_{\text{original}} \times S_{\text{net}}$$

  Before invoking `Socket.connect`, we first call `Thread.sleep(`$T_{\texttt{original}}$`-` $T_{\texttt{perturbed}}$`)` and then issue `connect` with the perturbed timeout $T_{\text{perturbed}}$. This additional change combines an artificial pre-delay with a shortened timeout window, keeping the total blocking within T_{original} while exercising slow-connection scenarios that can surface timing-sensitive test behavior as well. The strategy remains specification-compliant because the `Socket.connect` waits at most the requested timeout. Furthermore, we ensure the total blocking time never exceeds the original budgeted time. We also model name-resolution latency: in Java, host-name resolution is performed by `InetAddress` before the connect call via the OS resolver, so DNS time lies outside the connect-timeout budget and can add seconds of delay under adverse conditions; accordingly, we add a fixed, configurable pre-connect delay (default 1 second). Asynchronous or non-blocking networking stacks are out of scope for this work.

5 Experimental Setup

We answer the following research questions:

- **RQ1**: What is the overall effectiveness of ChaosAPI at detecting flaky tests?
- **RQ2**: What is the effectiveness of each strategy at detecting flaky tests?
- **RQ3**: How well does ChaosAPI generalize to new subjects?
- **RQ4**: How does ChaosAPI's cost-effectiveness compare to a rerun baseline?

We address RQ1 to assess whether ChaosAPI can systematically detect flaky tests. We evaluate its effectiveness by checking how well it detects known flaky tests from a prior dataset. We address RQ2 to measure the contribution of each strategy towards detecting flaky tests. We want to see which ones are more effective, in terms of detecting real flaky tests as well as their false positive rates, which in turn helps us understand better the root causes of flaky tests. We address RQ3 to examine how well ChaosAPI generalizes to a wider set of projects as well as whether it can detect new flaky tests within these different projects beyond the existing dataset. We address RQ4 to see whether ChaosAPI is cost-effective against a well-known and straightforward baseline of rerunning tests to detect flaky tests. If ChaosAPI can detect more flaky tests while taking the same or less amount of time, then ChaosAPI is practical.

Table 1. Evaluation corpus drawn from the FlakeFlagger dataset. This table defines stable project IDs (P1–P11) used across the paper.

ID	Project	Commit	# Tests	# Flaky Tests
P1	apache/commons-exec	ea4d1d4	55	1
P2	apache/httpcore	49247d2	707	22
P3	doanduyhai/Achilles	f52f7ec	1318	4
P4	elasticjob/elastic-job-lite	b022898	560	3
P5	jknack/handlebars.java	db77ef2	544	1
P6	joel-costigliola/assertj-core	22be382	6267	1
P7	ninjaframework/ninja	b1b58e8	307	1
P8	qos-ch/logback	0f57531	863	16
P9	tootallnate/java-websocket	fa3909c	146	21
P10	wro4j/wro4j	7e3801e	1163	15
P11	zxing/zxing	59ea393	345	2
—	sum	—	12275	87

5.1 Dataset

For our evaluation, we use Alshammari et al.'s FlakeFlagger dataset [2, 5], constructed by rerunning tests from open-source projects 10,000 times, recording any test that passes and fails across these reruns as flaky. In addition, they save the failure logs corresponding to the flaky failures. To evaluate ChaosAPI, we aim to run ChaosAPI on all tests within the projects defined with this dataset to see whether ChaosAPI can detect the same flaky tests.

We take additional steps to filter out projects from the dataset that are incompatible with our current implementation of ChaosAPI or our experiment goals. First, we exclude projects that failed to build with a standard Maven installation command (after removing various unnecessary checks for building, such as license checks), reducing the initial 25 projects to 16. Second, because our work targets flaky tests unrelated to test ordering, we ran the iDFlakies tool [17] that detects order-dependent flaky tests, whose outcomes depend solely on the order in which they are run, on the remaining projects, as done in prior work [32, 33]. We then remove any projects that only contain order-dependent flaky tests, resulting in 11 projects that remain. Within the FlakeFlagger dataset, these projects contain 87 total known flaky tests. We present in Table 1 the details about these projects, namely the total number of tests and the number of known flaky tests within these projects as labeled in the FlakeFlagger dataset.

5.2 Configuring Strategies

Each strategy has a configuration variable, as defined in Section 4, and we configure each strategy to use a specific value for that variable. Table 2 describes how we configure each strategy. We purposefully choose values that we believe to force out interesting edge cases or boundary conditions that are likely to induce failures from test executions. For example, for the Deterministic Value Strategy, we purposefully have the RNG APIs always return the maximum value possible (either the largest number possible when unbounded, or the largest value within bounds), as to force out test failures in case tests have some implicit assumption on the maximum bounds. We choose one such value for each strategy.

We evaluate ChaosAPI under two possible strategy use scenarios:

- **Combined mode.** We apply multiple strategies at once, perturbing many target APIs in one execution of the tests. This scenario aims to maximize the amount of perturbation as to more efficiently reveal flaky tests in just one execution. Because strategies within the same

Table 2. Perturbation strategy configurations used in our experiments.

ID	Group	Strategy	Configuration	Description
S1	Random	Deterministic Value	$V = MAX_VALUE$	Always return deterministic extreme values (e.g., `Integer.MAX_VALUE` or $n-1$ for `nextInt(n)`), exposing overflow and boundary-assumption bugs.
S2	Time	Fixed Increment	$\delta_{\text{time}} = 5000$	Advance time by 5000 each call, quickly driving executions past common timeout thresholds.
S3		Rounding	$G = 10000$	Round to 10000 granularity, making consecutive timestamps possibly equal and progression uneven, to stress assumptions of strict monotonicity and smooth advancement.
S4		Scaling (fast)	$S_{\text{fast}} = 10.0$	Scale time by 10× to stress time-sensitive assertions.
S5		Scaling (slow)	$S_{\text{slow}} = 10.0$	Scale time by 0.1× to emulate slow environments and reveal hidden assumptions on progress speed.
S6	Date	Fixed Moment	D_{target} = Jan 1, 3000	Choose year 3000 to test robustness of far-future logic handling and formatting (e.g., Year 2038 Problem).
S7	Locale	Locale Change	L_{target} = `tr_TR`	Switch to `tr_TR`, a locale with well-known casing quirks (e.g., dotted `i` vs. dotless İ), to stress formatting assumptions.
S8	Sleep	Sleep Increase	$I_{\text{sleep}} = 2000$	Add 2000ms delay to increase risk of timeouts and expose fragile waiting logic.
S9		Sleep Decrease	$D_{\text{sleep}} = 2000$	Subtract 2000ms to reduce wait times. Clamped to a minimum of 0.
S10	Concurrency	Timeout Shrinking	$S_{\text{con}} = 0.2$	Shrink timeouts by factor 0.2 to stress code that assumes generous blocking times.
S11	Network	Network Delay	$S_{\text{net}} = 0.01$	Shrink timeouts by factor 0.01 to emulate slow networks.

group target the same APIs, we can enable at most one strategy per group to avoid conflicts. Concretely, we select strategies **S1, S2, S6, S7, S8, S10, S11** (from Table 2) as the combined set. This scenario mirrors how practitioners can use ChaosAPI during regression testing: enable a small, diverse set of strategies to quickly detect flaky tests when the tests are run once after a change.

- **Single-Strategy mode.** We apply exactly one strategy at a time, and we execute the tests once per strategy, which allows us to isolate the effects of each strategy to flaky tests detected. While this scenario requires tests to be run multiple times, once per strategy, leading to higher

costs in using ChaosAPI, it allows developers to more precisely debug and isolate the cause for detected flaky tests, such as detected in Combined mode. We rely on Single-Strategy mode as part of our study in RQ2 to determine the effects of individual strategies on ChaosAPI effectiveness.

Additionally, we define a rerun baseline that simply reruns the test suite to detect flaky tests, representing a traditional detection approach. We configure rerun baseline to run the tests 30 times and report tests that failed as flaky tests. We choose this number of reruns as it is slightly over what we later found to be the time taken to run all strategies in single-strategy mode, allowing us to compare them as if they were given the same time budget to run.

5.3 Metrics

We first run all tests normally, without using ChaosAPI, to check that the tests indeed pass. Then, after running the tests through ChaosAPI, any test that fails is considered to be detected as a flaky test. Since we have a known set of flaky tests in our dataset, we can compare the flaky tests detected using ChaosAPI against those known ones to see whether ChaosAPI can detect those same flaky tests. We apply two different matching strategies to check whether a failing test is the same as one from the existing dataset. First, we match based on test name. If ChaosAPI can make a known flaky test fail, we consider it a match and consider ChaosAPI to detect that flaky test. However, it is possible that the failure resulting from using ChaosAPI is not the same failure as what was observed before from the many reruns. To avoid any risk that ChaosAPI leads to failures that are not possible from normal reruns, we perform a stricter matching that matches the failure log obtained from running through ChaosAPI against the known failure logs saved from the FlakeFlagger dataset, and we count a match when the ChaosAPI log matches any one of the recorded messages for that test. In this comparison, we ignore stack frames unrelated to the project itself such as those from JUnit, or Maven Surefire, and we tolerate minor variations in message details. More specifically, we treat the reports as equivalent as long as they originate from the same line number, share the same stack trace elements, and involve the same error type. We report results using both matching strategies. Any test not detected by ChaosAPI is then a negative. We can then compute false positives as flaky tests that ChaosAPI detects that are not in the dataset, and false negatives are the known flaky tests from the dataset that ChaosAPI does not detect. We can then compute standard precision/recall metrics based on these false positive/negative definitions.

We observe that while ChaosAPI may detect tests as flaky that were not in the original dataset, these tests may not all be false positives, since the FlakeFlagger dataset is not guaranteed to be comprehensive and contain all possible flaky tests within those projects. Since those flaky tests were identified solely through a large number of reruns without changing any environmental factors like time zone or locale, certain types of flaky tests whose flaky behavior comes from these differing environmental factors would not be found from simple reruns. For this reason, two of the authors manually inspected every failure reported by ChaosAPI to assess whether it represents a true flaky failure. As part of the validation, the authors inspected the error logs, test code, and the strategy used to induce the failure to decide whether it seems reasonable that the strategy could have made the failure happen and whether the test could have failed normally. In some cases, we made bigger changes to the runtime environment based on prior work and our own understandings of flaky tests to check for flaky failures. For example, we modified the locale on the operating system to see whether tests would still fail under different locales (instead of just perturbing the relevant API), and we injected delays in code to check whether timing-related failures could be reproduced, as proposed by prior work [18, 32].

Table 3. Evaluation corpus (open-source Java projects collected from GitHub).

ID	Project	Commit	# Tests
N1	alibaba/druid	a67023f	6015
N2	alibaba/fastjson	c942c83	5022
N3	Atmosphere/atmosphere	a6aef78	272
N4	awaitility/awaitility	9d1e0b5	170
N5	bytedeco/javacpp	b72bc6a	57
N6	changmingxie/tcc-transaction	874cb91	12
N7	coobird/thumbnailator	951d04f	2636
N8	DerekYRC/mini-spring	0a14861	34
N9	DiUS/java-faker	a8b8ff0	578
N10	FasterXML/jackson-databind	393fc3d	3457
N11	immutables/immutables	a020ddc	1274
N12	jhy/jsoup	bd155b8	1676
N13	johncarl81/parceler	52e6b14	88
N14	junit-team/junit4	84a9c46	1106
N15	justauth/JustAuth	694bbf1	52
N16	jwtk/jjwt	e3fff12	1725
N17	killme2008/aviatorscript	231198d	857
N18	knightliao/disconf	872e69c	37
N19	NanoHttpd/nanohttpd	efb2ebf	237
N20	opengoofy/hippo4j	7d78be3	420
N21	pagehelper/Mybatis-PageHelper	4db191c	139
N22	perwendel/spark	1973e40	318
N23	redis/lettuce	9ecee24	3334
N24	rest-assured/rest-assured	ef60d2c	794
—	sum	—	30310

We further assess the applicability of ChaosAPI when run on projects beyond those within the FlakeFlagger dataset for RQ3. We collected additional real-world projects from GitHub. Starting from the top 1000 Java repositories on GitHub based on star count, a representation of the popularity of GitHub projects, we collect the 351 projects that use Maven.

We executed `mvn test` in our default environment and excluded projects that (i) exceeded a pre-determined 10-minute per-run budget (necessary to allow many repeated trials under instrumentation without dominating wall-clock time), (ii) failed to build, or (iii) contained fewer than 10 tests, yielding a corpus of 42 projects. For presentation and evaluation focus, we report results only for 24 projects (out of 42) in which using ChaosAPI resulted in at least one test failure.

Table 3 lists these projects together with the commit SHA on which we ran the experiments and the number of tests at that commit. To inspect these flaky tests to determine true and false positives, we sampled projects in which failures were highly concentrated within a single strategy group, such that for a given group g, if a project P accounts for more than 30% of all failures attributed to g across all projects (i.e., $\text{fails}(P, g)/\sum_{P'} \text{fails}(P', g) > 0.30$), then we select P for manual inspection for this strategy group.

5.4 Hardware Configuration

We run all experiments on a machine configured with an Intel Core i9 9900 (16 cores/32 threads, 32 GB RAM), running in Ubuntu version 20.04.6 LTS, using JDK version 8u442 (Temurin) and Maven version 3.9.6.

Table 4. Evaluation summary with Combined mode and Single-Strategy mode.

ID	# RR–FT	Combined mode					Single-Strategy mode				
		# RF	# MP	# Both	# CP	# FFM	# RF	# MP	# Both	# CP	# FFM
P1	0	0	0	0	0	0	5	5	1	1	0
P2	0	33	10	17	4	2	38	15	19	6	6
P3	0	354	354	3	3	0	355	355	3	3	0
P4	0	3	3	2	2	2	4	4	3	3	3
P5	0	4	4	0	0	0	5	5	1	1	1
P6	1	26	24	1	1	1	26	24	1	1	1
P7	1	1	1	1	1	1	1	1	1	1	1
P8	0	50	48	6	6	5	54	52	9	9	8
P9	19	21	21	19	19	19	22	22	19	19	19
P10	7	8	8	5	5	5	18	18	15	15	15
P11	0	5	5	1	1	0	6	6	2	2	1
sum	28	505	478	55	42	35	534	507	74	61	55

6 Evaluation

6.1 RQ1: What is the overall effectiveness of ChaosAPI at detecting flaky tests?

Table 4 shows the aggregate results of running ChaosAPI using both combined mode and single-strategy mode on test suites from the FlakeFlagger dataset. The table also shows the results of flaky tests detected by simply rerunning the tests. "# RR–FT" is the number of flaky tests detected by rerun baseline. "# RF" is the total number of flaky tests reported by ChaosAPI (under different modes), namely the tests that failed when run. "# MP" is the number of flaky tests from "# RF" we manually inspected and confirmed to be true flaky tests. "# Both" is the number of flaky tests from "# RF" that intersect with known flaky tests in the FlakeFlagger dataset, matching based on test name. "# CP" is the number of flaky tests from "# MP" (the confirmed flaky tests based on manual inspection) that intersect with known flaky tests in the FlakeFlagger dataset, matching based on test name. Finally, "# FFM" shows the flaky tests detected using ChaosAPI whose failure message matches at least one failure message from the corresponding known flaky test from the FlakeFlagger dataset.

We can see from Table 4 that rerun baseline detects only 28 flaky tests, though all match the failure messages of corresponding flaky tests in the dataset. These tests are highly concentrated in tootallnate/java-websocket (P9), with 19 tests, and wro4j/wro4j (P10), with 7 tests. In contrast, running ChaosAPI in combined mode, even with a subset of the strategies and a single pass, reports many more flaky tests, overlapping with the reported ones from the FlakeFlagger dataset (55 tests vs. 28 tests), and 42 of them we confirmed to be true after manual inspection. In single-strategy mode, ChaosAPI detects at least one flaky test in every project, and identifies 74 name-matched tests, of which we confirm 61 to be true positives, yielding a recall of 70.1%.

ChaosAPI seemingly detects many more flaky tests than the number reported in the FlakeFlagger dataset, with 505 and 534 flaky tests detected in the combined mode and single-strategy mode, respectively. In our inspection of the failures to determine whether they are indicative of a true flaky test (Section 5.3), we confirm that 61 are true flaky tests, which were also reported in the FlakeFlagger dataset. In the FlakeFlagger dataset, beyond these flaky tests labeled in FlakeFlagger dataset, we validate an additional 446 detected tests to be true flaky tests. Based on this manual analysis, we find that ChaosAPI achieved a precision of 94.9% (where all manually confirmed flaky tests are considered as true positives). We label 27 out of 534 flaky tests as false positives, where 13

of them come from the FlakeFlagger dataset. We discuss these cases further in Section 6.2, when we discuss which strategies detected which flaky tests and why they would have false positives.

Note that single-strategy mode detects many more flaky tests than combined mode. Part of this gap comes from combined mode not enabling all strategies, so some flaky tests that fail using those excluded strategies only would naturally not be detected. Another part stems from cross–strategy interactions that can dampen the perturbation: for example, S11 (Network Delay Strategy) prolongs operations while S2 (Fixed Increment Strategy) advances the perceived clock, partially canceling the intended delay effect. As a result, 12 flaky tests were detected in single-strategy mode but not in combined mode. Conversely, a small number of tests that depend on the joint effect of multiple strategies may surface only in combined mode, as their failures require the combined perturbations to co-occur; we detect 2 flaky tests in combined mode but not in single-strategy mode.

To contextualize this performance, we compare ChaosAPI (specifically single-strategy mode, which achieved the highest detection rate) against the FlakeFlagger baseline. FlakeFlagger originally reports an overall precision and recall of 66% and 74% respectively across all projects in their paper [5]. However, after applying our consistent filtering rules (removing OD tests and excluding projects due to build failure) and using the original FlakeFlagger annotations as the ground truth, FlakeFlagger's precision and recall drop to 45.6% and 41.4%, respectively. Meanwhile, ChaosAPI achieves a precision and recall of 94.9% and 70.1%, respectively, on this same filtered set.

Importantly, this evaluation does not incorporate the newly identified and manually confirmed flaky tests discovered by ChaosAPI. When we expand the ground truth by adding these confirmed tests to the FlakeFlagger annotations, FlakeFlagger's precision and recall further both decrease to 6.1%, as most newly confirmed flaky tests were not found in their original experiment that reran tests 10,000 times. This difference is expected, as ChaosAPI simulates extreme but API-compliant perturbations and environment-dependent factors (e.g., locale and date), which are unlikely to be revealed by repeated executions under a fixed environment and time zone.

A Case Study on False Negatives. We observe that ChaosAPI is unable to detect 26 flaky tests labeled in the FlakeFlagger dataset, which we consider as false negatives (FN). There are also some tests which are confirmed as true positive but have a different cause of failure. For our manual inspection, we looked at one representative case from each project that had either a false negative or a mismatched failure cause.

From apache/httpcore (P2), we find one test that should fail due to issues in the environment, namely being unable to resolve a local host DNS, which is outside the control of the APIs we perturb. From doanduyhai/Achilles (P3), we find one test that could have failed if the time was perturbed more, so a difference in configuration value could have detected it. We found this similar issue for a flaky test in the project tootallnate/java-websocket (P9). For project apache/commons-exec (P1), we find a test where ChaosAPI could make the test fail, but we do not get the exact same recorded failure because the ChaosAPI-induced failure occurs earlier, stopping the test before it could move on to (potentially) fail for that later reason. For project zxing/zxing (P11), we find one test that should fail due to a difference in random numbers generated, but our strategy for RNG outputs the same number, forcing an infinite loop behavior, not matching the same recorded failure. A different strategy for RNG could potentially detect this flaky test. Finally, for qos-ch/logback (P8), we show in Fig. 4 the test where it depends on an email to be received in time. While we perturb timing-related APIs that could force a timeout early, the other email operation still persists in the background, unaffected by our perturbations. Even if we perturb to the point that there is no delay, that operation may still complete in time, making the test pass.

```
@Test
public void testCustomEvaluator() throws Exception {
    // ... configuration and logging calls omitted ...
    logger.debug("CustomEvaluator");
    // This method contains the waiting logic
    waitUntilEmailIsSent();
    // Verification fails if email is not received in time
    MimeMultipart mp = verifyAndExtractMimeMultipart(...); // Potential ERROR!!
    // ... assertions ...
}

// The core waiting logic that leads to the false negative
public boolean waitForIncomingEmail(long timeout, int emailCount)
        throws InterruptedException {

    WaitObject o = this.managers.getSmtpManager().createAndAddNewWaitObject(emailCount);
    // ...
    synchronized(o) {
        long t0 = System.currentTimeMillis();
        while(!o.isArrived()) {
            // Mechanism 1: Internal wait (Observation of state)
            o.wait(timeout);

            // Mechanism 2: Explicit timeout check (Observation of time)
             if (System.currentTimeMillis() - t0 > timeout) {
                return false; // The return value is not utilized.
            }
        }
        return true;
    }
}
```

Fig. 4. False Negative example from project qos-ch/logback.

Overall, we find that ChaosAPI remains effective at detecting flaky tests, in both single-strategy mode and combined mode, when evaluated against the known flaky tests in the FlakeFlagger dataset, finding more flaky tests than simply rerunning the test suite.

> **Answer to RQ1:** ChaosAPI achieves 70.1% recall on known flaky tests from the FlakeFlagger dataset when run in single-strategy mode that runs tests with each strategy individually. Furthermore, it also detects 446 new, previously unknown flaky tests that were not part of the previous FlakeFlagger dataset, with validated precision of 94.9%.

6.2 RQ2: What is the effectiveness of each strategy at detecting flaky tests?

To investigate the effectiveness of each of ChaosAPI's strategies, we show in Table 5 the breakdown of flaky tests detected in each project based on groups of strategies (Section 4). In the table, column "# RF" denotes the number of flaky tests reported when running strategies of each group, "# MP" denotes the number of those flaky tests confirmed as truly flaky through our manual inspection, and "# FFM" denotes the number of flaky tests whose failure message match those recorded in the FlakeFlagger dataset. We present analyses on the flaky tests detected by each group of strategies.

Table 5. Evaluation summary by failure category: Random, Date, Sleep, Time, Network, Locale, and Concurrency (per project; # RF/# MP/# FFM per group).

ID	Random			Time			Date			Locale			Sleep			Concurrency			Network		
	# RF	# MP	# FFM	# RF	# MP	# FFM	# RF	# MP	# FFM	# RF	# MP	# FFM	# RF	# MP	# FFM	# RF	# MP	# FFM	# RF	# MP	# FFM
P1	0	0	0	0	0	0	0	0	0	0	0	0	5	5	0	0	0	0	0	0	0
P2	0	0	0	34	11	3	0	0	0	0	0	0	5	5	2	2	2	2	3	3	0
P3	0	0	0	1	1	0	0	0	0	354	354	0	0	0	0	0	0	0	0	0	0
P4	0	0	0	1	1	1	2	2	2	0	0	0	1	1	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	0	0	0	4	4	0	1	1	1	0	0	0	0	0	0
P6	0	0	0	3	3	0	7	5	1	17	17	0	1	1	1	0	0	0	0	0	0
P7	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
P8	25	25	1	8	7	5	2	1	0	16	16	0	10	10	5	3	3	1	0	0	0
P9	3	3	3	2	2	1	4	4	4	3	3	3	4	4	4	1	1	1	20	20	18
P10	0	0	0	13	13	6	0	0	0	9	9	7	4	4	2	0	0	0	2	2	2
P11	5	5	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sum	33	33	4	64	40	18	16	13	8	403	403	10	31	31	15	6	6	4	25	25	20

Abbreviations: # RF = tests reported as failing by ChaosAPI; # MP = tests confirmed as flaky by manual validation; # FFM = tests whose identifiers and failure messages match the FlakeFlagger dataset.

Random. Flaky tests detected by strategies in the Random group are concentrated in a few projects. Our manual inspection shows these failures largely come from tests that implicitly rely on variability in RNG outputs. For example, in qos-ch/logback (P8), many tests repeatedly attempt to write to a temporary file whose name is derived from a random number, leading to unexpected behavior.

Time. The strategies in the Time group overall detect flaky tests that match those in the FlakeFlagger dataset across the most number of projects, within 9 of 11 projects. Many of these projects' tests rely on estimating or constraining execution time using `System.currentTimeMillis()` or `System.nanoTime()`, which are sensitive to machine speed and transient load.

However, these strategies also lead to the most false positives. Our manual validation on the false positives in apache/httpcore (P2) shows that these false positives are caused by an amplification effect: the Fixed Increment Time Strategy interacts with the project's custom NIO reactor, which performs timeout detection via a multi-threaded polling loop repeatedly invoking `System.currentTimeMillis()`. Because the perturbed time is a process-wide logical clock that advances on *each* read, concurrent polling causes increments to accumulate across threads, making the observed elapsed time grow with polling frequency/concurrency rather than wall-clock progress. This interaction amplifies the intended perturbation and leads the reactor to flag premature timeouts, resulting in what we consider as false positives. Notably, when a single control path measures elapsed time with occasional reads, the effect remains bounded; it is the tight, multi-threaded polling pattern that triggers this mismatch.

Date. Most flaky failures from the Date group of strategies arise from formatting and parsing issues. In joel-costigliola/assertj-core (P6), a test expects a fixed three-digit millisecond string. The test fails when it tries to get the current date, whose time part ends with one or two zeroes (e.g., ".010" or ".100") due to our perturbation. When the test calls `Timestamp.toString()` on this date, it trims the trailing zeros and prints ".01" or ".1", which fails the assertion on there being a three-digit millisecond string.

We also find tests in the project that make temporal assumptions: a test hard–codes an instant (e.g., `"2111-01-01"`) and asserts it must be strictly after "now". Under our Date perturbation, the perceived "now" can be any valid instant (e.g., a far future point), but the test implicitly assumes it to be near real time.

We observed three false positives from strategies of this group due to *inconsistency of the compared temporal values*: one operand is created through an instrumented path perturbed by ChaosAPI

(for example, constructed using `new Date()`), while the other is produced by an uninstrumented source such as parsing an XML file that got the time from an alternative time provider. Because only `new Date()` is perturbed, the two values diverge even though the code under test is correct. This partial-coverage mismatch is a limitation of our strategies only targeting specific APIs, as we may miss other related APIs, or the code relies on other sources to get the same information. This inconsistency is particularly noticeable for dates, because many tests cross-check project-specific time routines against Java Standard Library defaults.

For example, in a test in jknack/handlebars.java (P5), it obtains a value representing `now` via `Calendar.getInstance().getTime()`, which is unperturbed. Meanwhile, another call to method `apply()` that changes a `value` to `Date` triggers a direct `new Date()` call, which ChaosAPI perturbs. Because the two date values are constructed along different paths (unperturbed vs. perturbed), they diverge even though the program logic is correct.

Locale. The strategies from the Locale group detected flaky tests in over half of the projects while detecting the most flaky tests overall. Locale-sensitive behavior (e.g., case conversion, number/date formatting, parsing) is often buried in the core application logic, and a locale shift can break string comparisons or configuration parsing early and trigger cascading failures. This failure mode is largely missing from prior research and reports of flakiness, yet our results show its prevalence.

Since we purposefully choose the Turkish (`tr-TR`) locale that is relatively different in structure than other locales, the tests tend to fail under this unexpected runtime environment. The resulting positives are near-perfect: they align with known locale hazards, and we even cross-validated the correctness of these results by changing the environment locale of the entire machine, outside of ChaosAPI. Typical failures involve number/date formatting, case folding, and exact string matching used by reflection or SQL. For example, in doanduyhai/Achilles (P3), which had the most detected flaky tests, the metadata framework locates the getter for field `id` via reflection and constructs method names (e.g., `getId`) using the default locale `String.toUpperCase()`; under the Turkish locale, `id` becomes `İd`, so `getId` is not found and an exception is thrown.

Sleep. ChaosAPI detected flaky tests in 8 of 11 projects by the Sleep group strategies. Many projects use `Thread.sleep` in the main thread to "wait" for background work instead of proper synchronization (`wait`/`notify`, latches, futures). This pattern is brittle: whether the sleep is long enough depends on load and scheduling. In our experiments, Sleep group strategies did not introduce false positives, detecting flaky tests that rely on fixed delays for thread orderings.

Concurrency. The Concurrency group strategies detected relatively few flaky tests and showed no clear pattern across projects. While prior work has identified concurrency as a major source of flaky tests [23], our experiments suggest that the immediate cause here is not the related APIs that use timeouts, but the use of brittle scheduling mechanisms such as `sleep`, as discussed earlier.

Network. Flaky tests detected using Network group strategies are concentrated in a single project, tootallnate/java-websocket (P9). All errors originate from the same pattern: within a single test, `socket.connect(address, 2000)` is invoked twice, and the test is constrained by only a 5000ms time limit. As a result, even modest perturbations introduced by our strategy can easily accumulate and push the execution beyond the time limit, triggering spurious timeouts.

Some tests in wro4j/wro4j (P10) rely on reaching external endpoints (e.g., public websites) to verify connectivity. Such checks are highly environment-dependent, where the outcome depends on local network conditions and the external service's stability.

To better understand the relationships between strategy groups, we compute pairwise overlaps using the number of identical *(test, failure-message)* pairs of detected flaky tests by both groups. Fig. 5 shows the heat map representing these pairs.

The only clear association is between the Time and Sleep groups (12 shared failures). Both perturb the effective time limits set for waits and time-based assertions. Sleep adds delay to

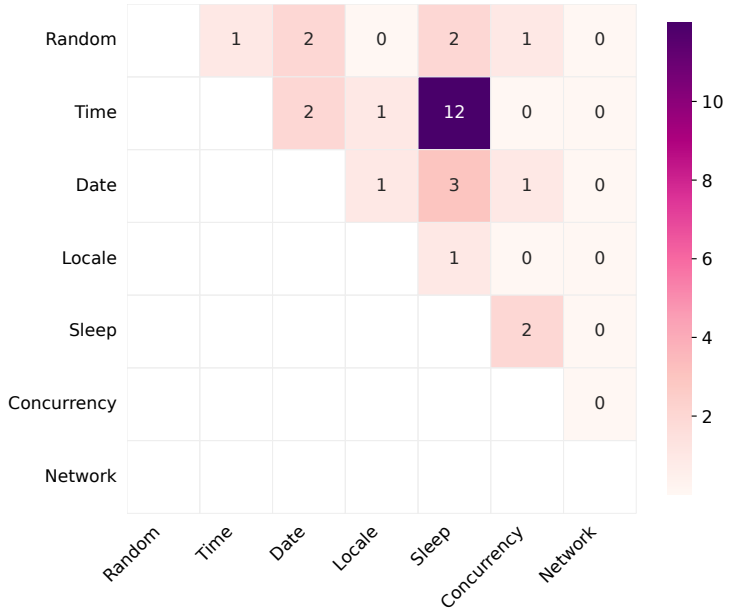


Fig. 5. Heat map of overlapping failures across different strategy groups.

foreground/background progress, while Time accelerates/rounds the clock. They often trigger the *same* timeout-style exceptions in the *same* tests.

Otherwise, most groups contribute largely *unique* failures. In particular, the Locale group yields 403 failures with only one overlap. From our inspection, these rerun baseline also detects these tests, so likely they do not fail due to Locale perturbation.

> **Answer to RQ2:** Time, Locale, and Sleep group strategies detect flaky tests across most projects, while Random, Date, and Network groups strategies detect flaky tests only among a few projects, as projects tend to not always use the related target APIs. These strategy groups are overall complementary where most detect unique flaky tests with unique failures.

6.3 RQ3: How well does ChaosAPI generalize to new subjects?

Table 6 summarizes the results of running ChaosAPI on new projects in single-strategy mode, shown as divided up among the strategy groups to compare with RQ2. ChaosAPI detected 40× more flaky tests compared to rerun baseline, detecting flaky tests in 24 projects while rerun baseline only detects flaky tests in 5 projects.

We observe similar patterns in number of flaky tests detected per strategy group as we found in RQ2. Locale detects flaky tests across many projects, but with heavy concentration within some individual projects, e.g. alibaba/druid (N1). Time similarly detects flaky tests among the largest number of projects while detecting the second most number of flaky tests overall. Random and Date groups' detected flaky tests are concentrated in specific projects, e.g., DiUS/java-faker (N9) for Random and jwtk/jjwt (N16) for Date. These distributions reinforce the generality of our observations across different projects.

Table 6. Evaluation summary by failure category (per project).

ID	# RR–FT	# RF	Random	Time	Date	Locale	Sleep	Concurrency	Network
N1	0	232	0	0	0	232	0	0	0
N2	0	19	1	0	2	16	0	0	0
N3	0	3	0	1	0	1	0	1	0
N4	0	54	1	48	1	0	15	1	1
N5	0	1	0	0	0	1	0	0	0
N6	0	2	0	0	1	0	2	0	0
N7	2	4	3	4	4	3	3	1	2
N8	0	1	0	0	0	0	0	0	1
N9	1	23	20	1	0	2	0	0	0
N10	0	9	0	0	0	9	0	0	0
N11	0	1	0	0	0	1	0	0	0
N12	0	15	9	1	0	0	0	0	6
N13	0	11	8	8	8	11	8	8	8
N14	0	17	2	12	2	7	4	2	2
N15	0	1	0	1	0	0	0	0	0
N16	0	20	0	0	20	0	0	0	0
N17	0	6	1	1	1	4	0	0	0
N18	0	3	0	0	0	3	0	0	0
N19	0	20	8	19	8	1	17	0	3
N20	4	3	0	0	1	0	2	0	0
N21	0	3	0	0	0	3	0	0	0
N22	4	17	4	16	0	0	2	0	0
N23	1	8	1	2	0	5	1	0	0
N24	0	9	2	9	2	2	2	2	2
sum	12	482	60	123	50	301	56	15	25

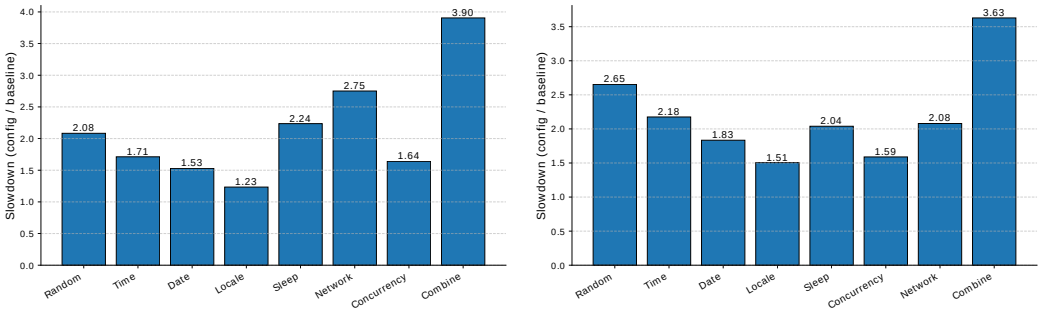
Abbreviations: # RR–FT = tests that failed during reruns; # RF = tests reported as failing by ChaosAPI; Random/Date/Sleep/Time/Network/Locale/Concurrency = counts per failure category.

We inspect flaky tests per group as described in Section 5.3. For alibaba/druid (N1), all Locale group flaky tests are confirmed as true flaky tests. The project contains many hard-coded SQL strings and assertions that implicitly assume English casing, thus locale-sensitive casing leaks into SQL generation/normalization methods (e.g., `String.format()` with the default locale), which are used to build or canonicalize keywords. This change in locale may corrupt some SQL tokens or keywords, e.g., `PRIMARY` becomes `PRİMARY`, which breaks exact-string assertions or parser expectations.

For DiUS/java-faker (N9), all Random group flaky tests are confirmed as true flaky tests. The library generates fake data via RNG-heavy sampling; several tests assume favorable draws or neglect boundary handling (Section 2). In one case, a test repeatedly samples until producing a valid SSN; under our perturbation, the loop fails to reach a valid state within a set time limit and instead times out.

For awaitility/awaitility (N4), all Time group flaky tests are confirmed as true flaky tests. Their failures primarily come from the Scaling (fast/slow) strategies. The most common pattern is that a test expects a task to satisfy a condition within a fixed window of time (e.g., one second), but the time scaling perturbs that time limit, causing a failure.

By contrast, we find that all Date group flaky tests for jwtk/jjwt (N16) are false positives. The cause matches our discussion on *inconsistency of the compared temporal values* in RQ2: one side uses a default `Date` (which we perturb), while the other is derived from unperturbed computations (e.g., a token's expiration date), leading to the mismatches.



(a) FlakeFlagger Dataset (b) New Projects

Fig. 6. Performance slowdown for each strategy group compared to baseline.

Answer to RQ3: On the new projects, ChaosAPI detects 40× more potential flaky tests compared to rerun baseline, demonstrating its effectiveness on new projects. The results share a similar distribution pattern of flaky tests detected per strategy as on the FlakeFlagger dataset, and we manually confirmed 300 as true flaky tests in a sampled subset.

6.4 RQ4: How does ChaosAPI's cost-effectiveness compare to a rerun baseline?

In Fig. 6, we report the runtime overhead of ChaosAPI when run in the combined mode as well as the overhead of each strategy group. Bars show the slowdown, measured as the ratio of the time it takes to run the tests after applying the strategies in that group to the time to run the tests once without any perturbation, averaged across projects. Fig. 6a summarizes the results on the FlakeFlagger dataset, while Fig. 6b summarizes the results on the new projects.

We observe that Locale strategy exhibits the smallest overhead on both datasets, as the perturbation is injected at a single callee site and requires minimal changes. In contrast, the combined mode run shows the largest cost ($\approx 3 \sim 4\times$), because multiple strategies are enabled concurrently and each loaded class is possibly instrumented several times. Nevertheless, its cost remains well below the sum of running the seven underlying strategies that form the combination, individually (e.g., 3.9× vs 14.0× in the FlakeFlagger dataset).

For the remaining groups, the slowdown typically clusters $\approx 2\times$ with project-to-project variance. Note that overhead is not only from bytecode instrumentation but also from intentional behavioral changes, e.g., changing network delay strategies can prolong blocking phases during execution. In practice, overhead correlates more with the frequency of target API invocations than with test-suite size. We expect ChaosAPI to be most cost-effective when applied selectively, for example by instrumenting only a subset of APIs, deploying a subset of perturbation strategies, or running it periodically (e.g., nightly or weekly) rather than on every CI commit. Importantly, even when summing the time of all single-strategy mode runs, the total cost (20.6× and 22.5× on the FlakeFlagger and new project datasets, respectively) is still below the 30× rerun baseline, while ChaosAPI detects substantially more flaky tests.

Answer to RQ4: ChaosAPI is more cost-effective compared to rerun baseline: Running ChaosAPI in both combined mode and single-strategy mode detects substantially more flaky tests with a reasonable overhead.

7 Discussion

Extensibility to Other Strategies. ChaosAPI is designed to be highly extensible: users can define new perturbation strategies by specifying how nondeterministic behavior is exposed through altered inputs or outputs at API boundaries.

In our study, target APIs were identified based on Java Standard Library documentation and prior knowledge of common sources of nondeterminism. This process could be further supported by automation, such as using LLMs to screen API documentation or program analysis to identify APIs whose behavior depends on some nondeterministic factor. Domain experts may also manually register project-specific APIs. As long as nondeterminism can be expressed via API-level perturbations, new strategies can be incorporated without changes to ChaosAPI's core design.

Extensibility to Other Language Systems. Although our implementation targets Java, the underlying methodology is not language-specific. Many languages expose analogous sources of nondeterminism through standard APIs, such as Python's `time.time()` [30], C/C++'s `rand()` [1], or Go's `net.DialTimeout()` [38]. Adapting ChaosAPI to other languages mainly requires replacing Java bytecode instrumentation with the corresponding instrumentation mechanism for that language, while preserving the same API-level perturbation principles. Perturbations should be guided by the official language specifications and standard library documentation, which define the permissible behaviors of these APIs.

Extensibility to Other Java Versions. We use JDK 8 to match the execution environment of FlakeFlagger. However, ChaosAPI is largely version-agnostic, as its perturbations are defined at the API level rather than relying on version-specific implementations. Extending ChaosAPI to other JDK versions mainly involves validating API semantics against official documentation and adjusting perturbations if specifications change. We also confirm that the documentation for our target APIs on the latest JDK versions remain the same, meaning our strategies should remain relevant on latest versions of Java.

Coverage of Nondeterministic Space. Exhaustively exploring the entire space of nondeterministic behaviors is infeasible. Our strategies therefore deliberately target representative edge-case behaviors, e.g., boundary values and specification-compliant corner cases, which are likely to expose latent assumptions in test code. Our manual inspections confirm that failures induced under these extreme scenarios correspond to flaky behaviors that could also manifest under more typical conditions. Developers with deeper domain knowledge can extend ChaosAPI by defining custom strategies with alternative perturbations, thereby broadening coverage to additional edge cases relevant to their codebase.

Guidance on Repair. The information reported by ChaosAPI can assist developers in diagnosing and repairing flaky tests. By reporting the API call sites whose perturbations trigger failures, ChaosAPI helps localize sources of nondeterminism. In our evaluation, we used this localization information during manual inspection when labeling false positives/negatives and analyzing newly discovered flaky tests. We expect API-level localization to be useful for future flaky-test repair tools, which could leverage it to narrow the repair search space. Moreover, the increased reproducibility enabled by controlled perturbations facilitates validation of candidate fixes, which is often challenging for flaky test failures.

8 Threats to Validity

Instability of flaky tests. By definition, flaky tests have nondeterministic outcomes, and their failures may not be completely reproducible across all runs. First, some tests already fail at a high rate even without using ChaosAPI, so they may coincidentally fail when run using ChaosAPI. We count such cases as true positives under our definition, but their exact messages/timings may

differ on reproduction. Second, several input-perturbation strategies (e.g., Network Delay Strategy) primarily *increase* failure likelihood rather than guarantee a failure. A latent flaky test may still pass when run under that strategy. More broadly, results can still be affected by environmental variance, even when applying perturbations.

Subjectivity in manual validation. We rely on human validation to confirm whether a failure is plausible under the perturbation and thus constitutes a true flaky test. Because our configurations deliberately stress edge conditions, people may disagree on what constitutes an "acceptable" scenario for a reported failure. Although we had two people review each flaky test and kept decision notes, errors remain possible (e.g., merging superficially similar messages that in fact reflect different root causes, or overlooking subtle mismatches).

Implementation-dependent effects. ChaosAPI is implemented as a Java agent based on the ByteBuddy framework [39]. In practice, agent-level rewriting can conflict with other bytecode tools or frameworks that also alter classes at load time, leading to occasional transform failures or missed hooks. Typical sources of interference include: (i) *mocking agents* that also attach early (e.g., Mockito [24] uses ByteBuddy) and (ii) *dynamic class generation* (e.g., Mockito/ByteBuddy, CGLIB-ASM based, Groovy-ASM based tools) that emits classes with unusual loader hierarchies or timing; these interactions can affect both coverage and observed overheads, and in rare cases induce false negatives or false positives by skipping intended transformations. We mitigate with explicit include/exclude matchers.

9 Related Work

Empirical studies have revealed the prevalence and impact of flaky tests across diverse domains. Luo et al. first categorized common root causes of flakiness, including asynchronous wait and concurrency issues [23]. Eck et al. studied flaky tests from the developer perspective, highlighting asynchronous wait, concurrency, and order-dependence as major categories [10]. Parry et al. also provided the first large-scale survey of flaky tests, synthesizing knowledge on their causes, consequences, detection, and mitigation strategies [27].

Building on these insights, researchers have developed a variety of techniques for detecting flaky tests. Alshammari et al. built a dataset of flaky tests found through rerunning tests 10,000 times to see which ones both pass and fail [5]. We use this dataset for our own evaluation. While effective in practice, rerun-based approaches are inefficient and may even fail to detect flaky tests that developers would later fix [26]. Researchers recently explored various ways of leveraging machine learning to predict which tests are flaky, leveraging either static features from parsing the code or dynamic features obtained from historical runs. Alshammari et al. trained on the dataset they created from rerunning tests 10,000 times to create FlakeFlagger, which predicts which tests are flaky using historic dynamic features [5]. Parry et al. proposed *CANNIER*, which combines re-execution with machine learning models to reduce detection cost [28]. Pontillo et al. explored static analysis and production-code features to predict flaky tests, showing that static metrics can perform comparably to dynamic baselines [29]. Fatima et al. fine-tuned a pre-trained large language model (LLM) to classify tests as flaky or not by only parsing test code [11]. Akli et al. introduced *FlakyCat*, a few-shot learning approach that categorizes flaky tests into types such as async waits, unordered collections, and time-related flakiness, though concurrency remains more difficult to detect [3]. Rahman et al. enhanced these prior LLM-based approaches by quantizing the models for better performance, then including a traditional random forest model as the final predictor to recover loss in accuracy [31]. While all this past work showed machine learning to be promising at detecting flaky tests, they ultimately may misclassify tests as flaky or not, due to the stochastic nature of machine learning itself. Our focus on running tests while perturbing APIs in controlled ways forces flaky tests to fail if they rely on these uncontrolled, nondeterministic components.

Beyond general-purpose detection, prior work developed techniques to detect or repair specific types of flaky tests. Lam et al. proposed iDFlakies to detect order-dependent flaky tests by changing the order in which tests are run [17]. Shi et al. followed up with iFixFlakies to repair order-dependent flaky tests [35]. Li et al. later proposed a different approach that leverages test generation techniques to similarly repair order-dependent flaky tests [21]. Rahman et al. developed FlakeRake to reproduce timing-dependent flaky tests through injected delays [32]. Later, Rahman and Shi developed FlakeSync to repair such tests by identifying where to synchronize test executions [33]. Chen and Jabbarvand proposed FlakyDoctor to repair flaky tests using LLMs [7], specifically targeting order-dependent and implementation-dependent flaky tests. Li et al. later developed FlakyGuard, a new LLM-based approach to repair flaky tests more generally, evaluated in an industrial setting at Uber [19]. We target different types of flaky tests than those handled by these past techniques, without relying on large language models. Our Time strategies may induce similar failures for timing-dependent flaky tests without injecting delays.

Existing mocking and time-faking techniques (e.g., libfaketime [12] or Java mocking frameworks such as Mockito [24] or JMockIt [22]) can simulate certain nondeterministic behaviors, typically at the unit-test or application level. ChaosAPI is complementary to these approaches: it perturbs nondeterministic APIs transparently at runtime, without requiring test rewriting or explicit mocks. This approach allows ChaosAPI to expose flaky behavior in unmodified test suites and across diverse sources of nondeterminism, while remaining lightweight to deploy.

Our work is most similar to prior work by Shi et al. that targeted APIs with nondeterministic specifications [34]. Their technique NonDex explores different orderings over data structures whose iteration order is not guaranteed, finding tests that can fail after running on different orderings. We similarly identify APIs that can have differing outputs and see whether tests can fail if we change their behaviors in acceptable ways, though we target a different set of APIs. We are also similar to work by Dutta et al. who changed the seeds to random number generators to see whether tests can fail with different sequences [9]. While we also target RNG APIs, our strategy is to have them always output some deterministic, edge-case value. Terragni et al. proposed a container-based infrastructure to support root-causing flaky tests using fuzzing-inspired techniques [37], which our approach is very similar to except we target specific APIs within the Java Standard Library.

10 Conclusions

We present ChaosAPI, a framework for detecting flaky tests by controlling the behavior of nondeterministic APIs. We define strategies that perturb the return values and input values of target APIs in a systematic manner, while remaining compliant to the specification of the API. We implement 11 strategies that target 17 different Java Standard Library APIs. We run ChaosAPI using these strategies on a dataset containing known flaky tests, demonstrating that ChaosAPI can detect most of these flaky tests with a recall rate of 70.1%. ChaosAPI also detected many other flaky tests that were not labeled as such in the original dataset, and our manual inspection confirmed most of them to be true positives, leading to a precision of 94.9%. We also evaluate ChaosAPI on new open-source projects beyond those in this prior dataset, detecting 300 flaky tests that we manually confirmed to be true positives, in contrast to the 12 detected through a simple rerun baseline. We also find using ChaosAPI to be more efficient at detecting flaky tests due to needing to only run tests once with the strategies, in contrast to rerun baseline, while still detecting more flaky tests.

Data-Availability Statement

We provide our implementation of ChaosAPI, together with experiment and data-analysis scripts and manually-labeled flaky results, at https://sites.google.com/view/chaosapi.

Acknowledgments

We thank the anonymous reviewers for their comments and feedback. This work was partially supported by NSF grants CCF-2145774 and CCF-2217696. We also acknowledge support from the Jarmon Innovation Fund.

References

[1] 2018. ISO/IEC 9899:2018 — Programming Languages — C.
[2] 2021. Flaky Test Dataset to Accompany "FlakeFlagger: Predicting Flakiness Without Rerunning Tests". https://zenodo.org/record/4450723#.YAhKgi1h1GQ.
[3] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2023. FlakyCat: Predicting Flaky Tests Categories using Few-Shot Learning. In *International Conference on Automation of Software Test*. 140–151. https://doi.org/10.1109/ast58925.2023.00018
[4] Nadia Alshahwan, Mark Harman, and Alexandru Marginean. 2023. Software Testing Research Challenges: An Industrial Perspective. In *International Conference on Software Testing, Verification, and Validation*. 1–10. https://doi.org/10.1109/icst57152.2023.00008
[5] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *International Conference on Software Engineering*. 1572–1584. https://doi.org/10.1109/icse43902.2021.00140
[6] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *International Conference on Software Engineering*. 433–444. https://doi.org/10.1145/3180155.3180164
[7] Yang Chen and Reyhaneh Jabbarvand. 2024. Neurosymbolic Repair of Test Flakiness. In *International Symposium on Software Testing and Analysis*. 1402–1414. https://doi.org/10.1145/3650212.3680369
[8] DiUS. 2025. java-faker. https://github.com/DiUS/java-faker. GitHub repository. Commit: `a8b8ff0acc6fcc629d08a3a9952f83be56a9a3c3`..
[9] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Misailovic Sasa. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *International Symposium on Software Testing and Analysis*. 211–224. https://doi.org/10.1145/3395363.3397366
[10] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840. https://doi.org/10.1145/3338906.3338945
[11] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. 2023. Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1912–1927. https://doi.org/10.1109/tse.2022.3201209
[12] Wolfgang Frisch. 2024. libfaketime: Fake time for user-space programs. https://github.com/wolfcw/libfaketime. Accessed Jan. 2026.
[13] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *International Working Conference on Source Code Analysis and Manipulation*. 1–23. https://doi.org/10.1109/scam.2018.00009
[14] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and Ranking Flaky Tests at Apple. In *International Conference on Software Engineering, Software Engineering in Practice*. 110–119. https://doi.org/10.1145/3377813.3381370
[15] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects Using continuous integration. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 821–830. https://doi.org/10.1145/3106237.3106288
[16] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *International Conference on Software Engineering*. 1471–1482. https://doi.org/10.1145/3377811.3381749
[17] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification, and Validation*. 312–322. https://doi.org/10.1109/icst.2019.00038
[18] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: Automated and Efficient Reproduction of Concurrency-Related Flaky Tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1520. https://doi.org/10.1145/3540250.3558956
[19] Chengpeng Li, Farnaz Behrang, August Shi, and Peng Liu. 2025. FlakyGuard: Automatically Fixing Flaky Tests at Industry Scale. In *International Conference on Automated Software Engineering*. 2158–2170. https://doi.org/10.1109/ase63991.2025.00179

[20] Chengpeng Li and August Shi. 2022. Evolution-Aware Detection of Order-Dependent Flaky Tests. In *International Symposium on Software Testing and Analysis*. 114–125. https://doi.org/10.1145/3533767.3534404
[21] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *International Conference on Software Engineering*. 1881–1892. https://doi.org/10.1145/3510003.3510173
[22] Rogério Liesenfeld. 2024. JMockit Toolkit. https://jmockit.github.io/. Accessed Jan. 2026.
[23] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering*. 643–653. https://doi.org/10.1145/2635868.2635920
[24] Mockito Contributors. 2024. Mockito Framework. https://site.mockito.org/. Accessed Jan. 2026.
[25] Oracle. [n. d.]. Java Platform SE 8: System.currentTimeMillis(). https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#currentTimeMillis. [Online; accessed 27-June-2025].
[26] Owain Parry, Michael Hilton, Gregory M. Kapfhammer, and Phil McMinn. 2022. What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection?. In *International Conference on Automation of Software Test*. 160–164. https://doi.org/10.1145/3524481.3527227
[27] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering Methodology* 31, 1 (2021), 1–74. https://doi.org/10.1145/3476105
[28] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2023. Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models. *Empirical Software Engineering Journal* 28, 72 (2023), 1–52. https://doi.org/10.1007/s10664-023-10307-w
[29] Valeria Pontillo, Fabio Palomba, and Filomena Ferrucci. 2022. Static Test Flakiness Prediction: How Far Can We Go? *Empirical Software Engineering Journal* 27, 187 (2022), 325–327. https://doi.org/10.1007/s10664-022-10227-1
[30] Python Software Foundation. 2024. The Python Standard Library. https://docs.python.org/3/library/. Accessed Jan. 2026.
[31] Shanto Rahman, Abdelrahman Baz, Sasa Misailovic, and August Shi. 2024. Quantizing Large-Language Models for Predicting Flaky Tests. In *International Conference on Software Testing, Verification, and Validation*. 93–104. https://doi.org/10.1109/icst60714.2024.00018
[32] Shanto Rahman, Aaron Massey, Wing Lam, August Shi, and Jonathan Bell. 2024. Automatically Reproducing Timing-Dependent Flaky-Test Failures. In *International Conference on Software Testing, Verification, and Validation*. 269–280. https://doi.org/10.1109/icst60714.2024.00032
[33] Shanto Rahman and August Shi. 2024. FlakeSync: Automatically Repairing Async Flaky Tests. In *International Conference on Software Engineering*. 1673–1684. https://doi.org/10.1145/3597503.3639115
[34] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *International Conference on Software Testing, Verification, and Validation*. 80–90. https://doi.org/10.1109/icst.2016.40
[35] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 545–555. https://doi.org/10.1145/3338906.3338925
[36] Denini Silva, Martin Gruber, Satyajit Gokhale, Ellen Arteca, Alexi Turcotte, Marcelo d'Amorim, Wing Lam, Stefan Winter, and Jonathan Bell. 2024. The effects of computational resources on flaky tests. *IEEE Transactions on Software Engineering* 50, 12 (2024), 3104–3121. https://doi.org/10.1109/icstw60967.2024.00027
[37] Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. 2020. A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests. In *International Conference on Software Engineering (New Ideas and Emerging Results Track)*. 69–72. https://doi.org/10.1145/3377816.3381742
[38] The Go Authors. 2024. Package net. https://pkg.go.dev/net. Accessed Jan. 2026.
[39] Rafael Winterhalter and contributors. 2025. *Byte Buddy: Runtime code generation for the JVM.* https://bytebuddy.net/ Version: 1.11.22; Accessed: 2025-10-11.
[40] Hengchen Yuan, Jiefang Lin, Wing Lam, and August Shi. 2024. Test scheduling across heterogeneous machines while balancing running time, price, and flakiness. In *International Conference on Software Maintenance and Evolution*. 449–460. https://doi.org/10.1109/icsme58944.2024.00048
[41] Celal Ziftci and Diego Cavalcanti. 2020. De-Flake Your Tests: Automatically Locating Root Causes of Flaky Tests in Code At Google. In *International Conference on Software Maintenance and Evolution*. 736–745. https://doi.org/10.1109/icsme46990.2020.00083

Received 2025-10-10; accepted 2026-02-17