Fray: An Efficient General-Purpose Concurrency Testing Platform for the JVM

AO LI, Carnegie Mellon University, USA
BYEONGJEE KANG, Carnegie Mellon University, USA
VASUDEV VIKRAM, Carnegie Mellon University, USA
ISABELLA LAYBOURN, Carnegie Mellon University, USA
SAMVID DHARANIKOTA, Carnegie Mellon University, USA
SHREY TIWARI, Carnegie Mellon University, USA
ROHAN PADHYE, Carnegie Mellon University, USA

Concurrency bugs are hard to discover and reproduce, even in well-synchronized programs that are free of data races. Thankfully, prior work on controlled concurrency testing (CCT) has developed sophisticated algorithms—such as partial-order based and selectively uniform sampling—to effectively search over the space of thread interleavings. Unfortunately, in practice, these techniques cannot easily be applied to real-world Java programs due to the difficulties of controlling concurrency in the presence of the managed runtime and complex synchronization primitives. So, mature Java projects that make heavy use of concurrency still rely on naive repeated stress testing in a loop. In this paper, we take a first-principles approach for elucidating the requirements and design space to enable CCT on arbitrary real-world JVM applications. We identify practical challenges with classical design choices described in prior work—such as concurrency mocking, VM hacking, and OS-level scheduling—that affect bug-finding effectiveness and/or the scope of target applications that can be easily supported.

Based on these insights, we present *Fray*, a new platform for performing push-button concurrency testing (beyond data races) of JVM programs. The key design principle behind Fray is to orchestrate thread interleavings without replacing existing concurrency primitives, using a concurrency control mechanism called *shadow locking* for faithfully expressing the set of all possible program behaviors. With full concurrency control, Fray can test applications using a number of search algorithms from a simple random walk to sophisticated techniques like PCT, POS, and SURW. In an empirical evaluation on 53 benchmark programs with known bugs (SCTBench and JaConTeBe), Fray with random walk finds 70% more bugs than JPF and 77% more bugs than RR's chaos mode. We also demonstrate Fray's push-button applicability on 2,664 tests from Apache Kafka, Lucene, and Google Guava. In these mature projects, Fray successfully discovered 18 real-world concurrency bugs that can cause 371 of the existing tests to fail under specific interleavings.

We believe that Fray serves as a bridge between classical academic research and industrial practice—empowering developers with advanced concurrency testing algorithms that demonstrably uncover more bugs, while simultaneously providing researchers a platform for large-scale evaluation of search techniques.

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

Additional Key Words and Phrases: Controlled Concurrency Testing, Debugging, JVM

Authors' Contact Information: Ao Li, aoli@cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Byeongjee Kang, byeongjee@cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Vasudev Vikram, vasumv@cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Isabella Laybourn, ilaybour@alumni.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Samvid Dharanikota, sdharani@alumni.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Shrey Tiwari, shrey@cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Rohan Padhye, rohanpadhye@cmu.edu, Carnegie Mellon University, Pittsburgh, USA.

ACM 2475-1421/2025/10-ART417
https://doi.org/10.1145/3764119

ACM Reference Format:
Ao Li, Byeongjee Kang, Vasudev Vikram, Isabella Laybourn, Samvid Dharanikota, Shrey Tiwari, and Rohan Padhye. 2025. Fray: An Efficient General-Purpose Concurrency Testing Platform for the JVM. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 417 (October 2025), 28 pages. https://doi.org/10.1145/3764119

1 Introduction

Software testing is the predominant form of validating correctness for large real-world programs due to its simplicity, wide-spread applicability, efficiency, and reproducibility. However, testing *multi-threaded* programs remains challenging in practice, despite the fact that concurrency bugs are among the most difficult to detect and diagnose [42, 45].

Take Java, which by several metrics is the most popular programming language with native support for concurrent multi-threading [17, 64, 67]. Let's assume that programmers write test cases to validate the correctness of concurrent programs via assertions (e.g., that a *parallel-sort* operation produces a sorted list). How can we check such properties? One might assume that developers can leverage 20+ years of academic research on concurrency testing to run state-of-the-art techniques; however, the state-of-the-practice is simply re-running concurrent tests multiple times to check whether some assertion fails [2, 50]. This approach is neither effective nor reproducible—for example, Apache Kafka's issue repository is teeming with discussions on concurrency-induced flaky tests as well as hard-to-replicate production failures [10]. What's missing here?

Let's step back and consider how a concurrency testing tool could help. Ideally, such a system would (a) provide an efficient mechanism for *controlled concurrency* [66]; that is, to deterministically execute a multi-threaded program along a fixed *schedule* (i.e., sequence of thread interleavings), (b) provide support for systematically or randomly exploring thread schedules using state-of-the-art search strategies (e.g., *partial order sampling* [75]) to uncover hard-to-find concurrency bugs, and (c) target any multi-threaded program without requiring much manual effort (e.g., in rewriting application code or using specialized testing DSLs). Unfortunately, no such general testing framework for the JVM currently exists, despite decades of research on concurrency testing *algorithms*. Why is that?

To understand this gap, we first look to systems described in prior work and map the design space, identifying the subtle trade-offs which make concurrency control of JVM programs challenging. Classical design choices include intercepting OS-level thread-scheduling decisions (as in RR [46], the record-and-replay tool for Linux), by emulating the JVM completely (as in Java Path Finder (JPF) [69]), by mocking concurrency primitives in the JDK (as done by Lincheck [33], for testing concurrent data structures), or by pausing individual threads to defer their scheduling (as in CalFuzzer [31]). As we will detail in Section 3, these design choices have impacts on (i) the *scope of what can be tested*, owing to *applicability* (i.e., the extent to which arbitrary target programs are supported), the *expressibility* of the search space (i.e., whether all interleavings can be deliberately and faithfully exercised), and the *maintainbility* of the tool itself (i.e., how easily it can keep up with evolving Java versions); as well as (ii) its *bug-finding effectiveness*, which for a given search algorithm (e.g., random walk) depends on maximizing the run-time *performance* of executions and minimizing the *search space* of which interleavings need to be considered.

Crucially, we note that the vast majority of prior academic work in this area has primarily focused on evaluating specific testing techniques or search algorithms, not on maximizing the practical applicability of their artifacts, which has historically been overlooked as an engineering concern. While understandable from a scientific point of view, a side effect is that the gap between research and practice is wider than ever before, with systems presented at OOPSLA 2024 [58] being limited to Java versions that were superceded in 2017. In contrast, this paper explicitly investigates the research question: "*Why is concurrency control with managed runtimes challenging, and what are the fundamental system design requirements for maximizing applicability to aribtrary JVM targets?*"

In order to make controlled concurrency testing effective and practically viable for managed code, the key design philosophy we establish is to orchestrate thread interleavings (i) without replacing existing concurrency primitives with mocks, while (ii) still encoding the semantics of these concurrency primitives for faithfully expressing the set of all possible program behaviors.

In this paper, we present *Fray*, the first concurrency testing platform for the JVM designed explicitly to maximize both general-purpose applicability as well as application-level bug-finding efficiency, while also providing correctness guarantees and a framework for extensibility. Fray's objective is to find concurrency-induced assertion violations, run-time exceptions, as well as deadlocks, in programs that are otherwise testable.[1] To perform concurrency control, Fray introduces a mechanism called *shadow locking*, which mediates access to shared resources in a specified order (the "thread schedule") with extra locks whose semantics are coupled to concurrency primitives used in the original program. Fray works on off-the-shelf JVM programs (compiled from Java, Scala, Kotlin, etc.), requiring no manual annotation or source rewriting. Fray efficiently performs deterministic concurrency control over the space of application-thread interleavings where context switches occur only at synchronization points. Fray's search space is *sound* and—in the absence of data races and other sources of non-determinism (e.g., timers)—also *complete*; that is, every concurrency bug discovered by Fray will be a true positive, and for every concurrency bug that can manifest in the original program there is a corresponding interleaving that Fray can execute to uncover it. In order to search for concurrency bugs, Fray can run various well-known algorithms such as random walk, probabilistic concurrency testing (PCT) [4], and partial-order sampling (POS) [75]. For debugging purposes, any saved interleaving can be deterministically replayed as long as the program does not depend on other sources of non-determinism.

We empirically show that, as a platform, Fray outperforms currently available alternatives for performing controlled random scheduling—RR and JPF—in bug-finding effectiveness on 53 programs from independently developed benchmark suites JaConTeBe [39] and SCTBench [66] (the latter ported to Java). We also demonstrate Fray's push-button applicability to 2,664 concurrent test cases from mature software projects such as Apache Kafka [16], Apache Lucene [15], and Google Guava [19]—we believe this is the *largest evaluation of controlled concurrency testing on real-world software*. Fray has successfully identified 18 distinct concurrency bugs across these projects (16 confirmed and 12 fixed so far), including both *previously unknown* bugs as well as *known* bugs that the developers could not previously reproduce for debugging.

Fray benefits both practitioners as well as the research community. For example, *Elastic Search Labs* published a blog post about how Fray helped them diagnose and fix tricky concurrency bugs in Lucene [34]. These bugs were exposed by running existing off-the-shelf unit tests with sophisticated partial-order sampling (POS) [75], which to our knowledge has not been applied at this scale before. Further, we were easily able to implement the bleeding-edge SURW algorithm [76] in Fray and provide a complementary evaluation on thousands of test targets.

To summarize, the contributions of this paper include:

(1) We elucidate the requirements (Section 2) and map the design trade-off space (Section 3) for performing practical concurrency control for programs running within the managed environment of a JVM.

[1]That is, the programs have entry points or test harnesses for executing logic that does not heavily depend on external sources of non-determinism such as randomness, timing, or networked I/O.

(2) We present Fray, a new platform for performing efficient concurrency control of JVM programs. We describe Fray's key design choices involving *shadow locking* (Section 4) and make the implementation available at https://github.com/cmu-pasta/fray.
(3) We empirically evaluate Fray by comparing to RR and JPF on concurrency-bug benchmarks from prior work, as well as on 2,600+ off-the-shelf tests from real-world Java software (Section 5)—the results demonstrate Fray's advantages in terms of performance, bug-finding effectiveness, and push-button applicability.
(4) Fray provides a bridge across academic research and industrial practice, enabling researchers to evaluate advanced concurrency testing algorithms on real-world JVM applications while providing practitioners access to state-of-the-art concurrency testing techniques.

2 Problem Definition

2.1 Problem Scope

Our goal is to find *concurrency bugs* in Java-like programs that perform multi-threading, where the bug is identified by an assertion violation, run-time exception, or a deadlock which only manifests under certain interleavings of threads—that is, the bug is induced by a *race condition*. This use of the term *concurrency bug* follows the style of the seminal work by Lu et al. [42], which included atomicity violations, ordering violations, deadlocks, etc. but not *data races*. This nuance is subtle but important.

In Java, the term *data race* refers to concurrent conflicting accesses (i.e., one write and another read/write) to a non-volatile shared variable. Java's weak memory model allows programs with data races to exhibit behavior that cannot be explained by *any* sequence of thread interleavings [21]. Data races can be effectively identified by race detectors [12, 13, 47, 60, 62] and can be easily fixed by using proper synchronization to control access to shared memory [40]. For our purposes, we assume (but not require) that programs are *free of data races* since developers can use the aforementioned techniques to remove them. Data-race-free Java programs exhibit sequential consistency [21]; so, we can explain concurrency bugs to developers by demonstrating a specific sequence of thread interleavings called a *schedule*.

Figure 1 depicts a sample (data-race-free) Java program which spawns two threads (Line 18), one of which sets local `x=0` and the other sets local `x=1` (Line 6). The program often terminates successfully; however, it can non-deterministically deadlock or trigger an assertion violation. For example, if the thread that calls `notify()` (Line 11) enters the `synchronized` block (Line 7) before the thread that calls `wait()` (Line 9), then the second thread will wait forever in a deadlock. Otherwise, if the thread that calls `notify()` enters the `synchronized` block second and then updates the shared `volatile` variable `b` (Line 14) before the thread waking from `wait()` can do so, then the value of `b` will end up being `0`, triggering an assertion failure (Line 20).

2.2 Concrete Objectives

We list three concrete objectives (O1–O3) for designing an ideal controlled concurrency testing platform for the JVM.

O1: Real-World Push-Button Testing: We want a concurrency testing system that can run on off-the-shelf JVM programs. This means (a) targeting general-purpose concurrency bugs that violate arbitrary program assertions, instead of only checking specific properties such as linearizability [23, 33] or class thread-safety [38]; (b) eliminating the need for developers to manually set up concurrency testing, such as writing specifications in a DSL, rewriting source code to use mocked concurrency libraries, or implementing support for specific frameworks—ideally, we want a drop-in wrapper for the `java` command; and (c) being able to run on mature real-world software—the

```java
class Foo extends Thread {
  static Object o = new Object();
  static AtomicInteger a = AtomicInteger();
  static volatile int b;
  public void run() {
    int x = a.getAndIncrement();
    synchronized(o) {
      if (x == 0) {
        o.wait();
      } else {
        o.notify();
      }
    }
    b = x;
  }
  public static void main(...) {
    Foo[] threads = {new Foo(), new Foo()};
    for (var thread : threads) thread.start();
    for (var thread : threads) thread.join();
    assert (b == 1);
  }
}
```

Fig. 1. Sample Java program containing several concurrency primitives. The program is well synchronized (i.e., no data races) but has concurrency bugs: it can non-deterministically run to completion, deadlock, or trigger an assertion violation.

latter requires designing a system such that the engineering effort (on the tool maintainer's part) to support a variety of application uses and keep up-to-date with evolving JDK versions is minimized in practice.

O2: Deterministic and Faithful Concurrency Control: We want to be able to provide a *thread schedule* during program execution such that it exhibits a specific sequence of thread interleavings; this should allow perfect replay debugging if the only source of non-determinism in a program is its concurrency. We also want to be able to control the thread schedule so as to employ randomized or systematic search strategies for testing. Ideally, we want to ensure (a) *soundness of expressibility*: the program behavior observed via any runnable thread schedule in the controlled testing system should correspond to behavior that can manifest in the original program. (b) *completeness of expressibility*: any concurrency-induced behavior from the original program can manifest with some expressible thread schedule in the controlled testing system.

O3: Support for Efficient Search-Based Testing: In order to achieve efficient concurrency testing for JVM programs, we need to achieve two goals: *performance* and *search-space optimization*. First, we want to minimize run-time overhead when deterministically executing a program along a fixed thread schedule. Typically, this means executing at least the non-synchronizing program instructions (e.g., memory reads/writes) at the same speed as during normal execution. Note that executing a multi-threaded program by scheduling threads one at a time sequentially is acceptable; during concurrency testing, we can parallelize the search algorithms by running different schedules across available CPUs, which maintains overall testing throughput (i.e., execs/time). Second, we want to minimize the search space for concurrency testing by (a) abstracting away the non-determinism within the managed runtime (e.g., VM initialization, garbage collection, class loading), and (b) only considering thread interleavings at synchronization points, which is sufficient for data-race-free programs (ref. Section 4.2).

Table 1. Summary of design choices and trade-offs for implementing concurrency control in the JVM. For each design choice, the second column lists the impacted attributes (and corresponding objectives from Section 2.2 in paranthesis).

Design Choice	**Impact (Objectives)**	**Examples (Prior and Proposed Work)**
OS-level interception	Search Space (O3), Performance (O3)	RR [48] and CHESS [45] intercept system calls, but cannot distinguish between concurrency in Java application threads vs. JVM internals, and also cannot search over thread schedules without restarting the JVM.
VM Hacking	Performance (O3), Applicability (O1), Maintainability (O1)	JPF [69] emulates the JVM, losing out on JIT optimizations and needing stubs for all native JDK methods needed for execution. DeJaVu [6] (Java 1.0) was a fork of the Sun JVM that did not keep up with Java versions.
Concurrency Mocking	Scope (O1), Maintainability (O1)	Lincheck [33] replaces synchronization primitives with custom implementations, but is primarily designed for testing data-structure linearizability. JESS [65] (Java 7) and JMVx [58] (Java 8) use concurrency mocking for general-purpose control, but cannot interoperate with the JVM's use of concurrency primitives in modern Java versions.
Directed Yielding	Applicability (O1), Determinism (O2), Expressibility (O2)	CalFuzzer [31] (Java 6), IMUnit [25] (Java 6), and Thread-Weaver [20] (Java 6) orchestrate thread execution by yielding / blocking based on external hints or specifications about event ordering. Given the relatively simple control mechanism, they cannot express or deterministically induce certain interleavings with `wait` and `notify`. IMUnit can also produce spurious deadlocks not present in the original program.
Shadow Locking (Our design)	-	***Fray***'s design enables push-button testing of arbitrary JVM programs, provides guarantees of deterministic faithful control for all data-race-free programs, and is effective at finding concurrency testing due to its optimized search space and high run-time performance.

3 Design Space and Related Work

In this section, we walk through various design choices encountered in implementing a concurrency control platform intended to meet objectives O1–O3. In doing so, we discuss systems described in prior work, how their designs lead to trade-offs across various quality attributes that ultimately affect one or more of our stated objectives, and what design choices Fray makes in response. Table 1 summarizes this discussion.

OS-level Interception: At one extreme, system-level thread scheduling decisions can be recorded and/or manipulated by intercepting system calls. For example, RR [48] was originally designed for record-and-replay of Linux programs, but it can be used for random concurrency testing via its *chaos mode* [46] which shuffles CPU priorities during replay. CHESS [45] provides systematic concurrency control for Windows programs by intercepting calls to Win32 concurrency APIs; it supports search strategies such as iterative context bounding [44]. However, for programs running in a managed runtime like the JVM, a system-level approach to concurrency control captures far more scheduling decisions than necessary—for example, the non-determinism within JVM initialization, garbage collection, and class-loading, neither of which affect program semantics—leading to a bloated *search space* (ref. Section 5 for an empirical evaluation with RR). Ideally, we would want a platform that can distinguish application threads from JVM internals.

D1: For Fray, we made the design decision to control only application-level concurrency and ignore the concurrency within the JVM.

VM Hacking: At the other extreme end, the Java Virtual Machine itself can be modified or replaced in order to take full control of concurrent execution semantics. The most prominent example of this approach is Java Path Finder (JPF) [69], which was originally designed for model checking but can also be used for random state-space exploration. JPF simulates program execution using a custom bytecode interpreter. This gives JPF full control over program execution semantics but it affects *performance* because it cannot use run-time optimizations (e.g., the HotSpot JIT Compiler). Moreover, JPF requires hand-written implementations of native JDK library methods, limiting *applicability* when certain programs depend on classes whose native methods have not been modeled, while also increasing the cost of tool *maintainability* as the JDK evolves (ref. Section 5 for an empirical evaluation). An older system, DejaVu [6], modified the Sun JVM (Java 1.0) to force determinism; unsurprisingly, the implementation has not kept up with modern Java.

D2: For Fray, we made the design decision to instrument JVM bytecode so as to run on existing production JVMs.

Concurrency Mocking: A popular middle ground to concurrency control is to simply substitute language-level concurrency APIs with mocks, either via source rewriting or IR instrumentation. The mocks provide applications the familiar threading and synchronization interfaces, but under the hood they can avoid using the native concurrency primitives completely and instead schedule application-level tasks in a controlled manner. This approach works really well for languages without managed runtimes. For example, Shuttle [1] and Loom [68] use concurrency mocking for testing Rust programs. Kendo [49] and DThreads [41] provide deterministic multi-threading for C/C++ programs by replacing `pthreads`.

However, for managed runtimes like the JVM, the concurrency mocking approach runs into limitations. The main challenge is in dealing with interactions between application code under concurrency control (e.g., a Java program), and the managed runtime, which is not (e.g., native C++ code within a JVM). A tool that replaces concurrency primitives in application code with mocks inevitably runs into issues when the same primitives are manipulated by the JVM, in a way that cannot be mocked. For example, consider the following facts:

- Java implements the `synchronized` keyword using locks on (hidden) object monitors [22], which must follow block-structured access (i.e., matching acquires/releases within the same method body).
- The thread co-operation instructions `wait()` and `notify()` respectively release and acquire nested monitor locks *implicitly* (i.e., the JVM does this automatically) [22].
- When application code calls `Thread.join()`, it uses `wait()` internally; when that thread terminates, the JVM *implicitly* calls `notifyAll()` to wake up joiners [52].
- The JVM can *implicitly* lock a class loader if it is not registered as parallel capable [51], but it is also common for application code to synchronize on class loader objects *explicitly* [14].

Taken together, these facts have several complex implications for any tool that performs concurrency mocking. For example, if a tool replaces `wait` and `notify` methods with custom mocks, then it must also replace all use of `synchronized` blocks since the mock methods cannot acquire/release the original monitor locks given Java's block-structuring requirements. But using mocks for monitors (i.e., mocking `synchronized`) means that there is no longer mutual exclusion during concurrent class loading or initialization when the JVM performs locking on the original monitors of the class loader. And now that `Thread.join()` calls a mocked `wait()`, it cannot observe thread termination when the JVM implicitly issues a `notifyAll()` using the original (non-mocked) signal. Each such interaction needs special handling. When considering thread pools, futures, etc. the list of workarounds needed goes on even longer. In practice, existing mocking-based tools are explicitly or implicitly restricted in *applicability*, as follows.

Lincheck [33] uses JVM bytecode instrumentation to replace a subset of synchronization primitives with custom implementations for testing the linearizability [23] of concurrent data structures [28]. Lincheck requires an annotated list of data-structure APIs and then it systematically invokes these methods concurrently from a fixed set of custom thread instances. Since Lincheck creates and manages the application threads itself in this use case, it can better control the surface of these application–JVM interactions.

JMVx [58], a record-and-replay system for the JVM, controls the non-determinism of thread interleaving by utilizing `Unsafe` APIs to modify concurrency primitives directly. However, this reliance on `Unsafe` APIs limits its *applicability* to Java 8, as these APIs were deprecated and subsequently removed in later Java versions (circa 2017). For record/replay, JMVx must also deal a host of other *maintainbility* challenges to adapt to newer versions of Java (see [58], §3.8–§3.9).

JESS [65], a Java port of CHESS, employs a more aggressive approach by replacing all concurrency primitives. It implements *method doubling* to avoid interference with the JVM runtime, ensuring that only application threads run the instrumented code (and not, for example, the classloader). While innovative, this strategy inherently limits its *applicability*, as it cannot handle class constructors, reflection, and virtual methods. These limitations frequently result in VM crashes and unpredictable behavior (see [65], §5.2).

D3: For Fray, we made the design decision to *avoid mocking any existing concurrency primitives* so as to streamline our engineering effort.

Directed Yielding: An alternative to concurrency mocking is to force threads to co-operatively yield execution to a special scheduler at key points (e.g., at the end of a critical section). This strategy is commonly used in tools where some information about interesting interleavings is provided via an external input, such that the scheduler can decide which thread to suspend and which to resume. For example, CalFuzzer [31] is an "active testing" framework for Java 6 that supports various concurrency testing algorithms [32, 56, 59, 60]. Active testing relies on inputs from an initial (imprecise) analysis phase that identifies potentially buggy locations. IMUnit [25] and Thread-Weaver [20] provide a framework for unit testing of multi-threaded Java code by explicitly specifying event orderings.

However, there are several limitations with this approach. Directed yielding tools are restricted in *applicability* since they are not push-button. Extending their control mechanisms for systematic testing is not straightforward, due to the complex semantics of primitives like monitors and signals. For example, IMUnit can introduce spurious deadlocks if an infeasible set of orderings is provided [25], sacrificing *soundness of expressibility*. CalFuzzer (made for Java 6) attempts to avoid this issue by using a now-unsupported `Unsafe` API to query the status of monitors [61], but this is not possible with modern Java. When dealing with Java programs that use `wait` and `notify`, none of these tools can *deterministically* control which one of multiple waiting threads wakes up when a `notify` signal is received. Similarly, some interleavings such as a "delayed wake-up" (that is, when a notifying thread releases and re-acquires a monitor lock before a waiting thread is woken up) are impossible to simulate in all of these tools, thus affecting the *completeness of expressibility*.

D4: For Fray, we made the design decision to explicitly encode the semantics of existing concurrency primitives in our thread control mechanism, so as to guarantee soundness and completeness of expressibility.

3.1 Other Related Work

LEAP [24], ORDER [72], and CARE [29] enable record-replay-debugging of concurrent Java executions via object-centric logging. However, they cannot be used to pick specific thread schedules for concurrency testing.

Researchers have extensively studied the problem of *flaky tests* [3, 35, 57] that pass or fail non-deterministically. Shaker [63] specifically targets flaky test detection due to concurrency issues, but it does not have any mechanism for controlling the non-determinism either for a systematic search or for deterministic replay.

Coyote [8] is a concurrency testing framework for C# programs using the Task Asynchronous Programming (TAP) model. It uses source or binary rewriting to replace concurrency primitives with custom mocks, similar to Lincheck and JESS. Likely due to similar complexities related to interoperability between applications and the runtime, Coyote does not control concurrency for C# programs that use bare threads instead of the TAP model.

Razzer [27], SnowCat [18], Ozz [26], DDRace [74], Conzzer [30], MUZZ [5], and RFF [71] use various forms of fuzzing to find data races, deadlocks, or concurrency-related program crashes in C/C++ programs or the Linux kernel. Periodical scheduling [70] parallelizes threads in defined periods instead of searching over fixed sequences of interleavings—this is achieved through Linux's deadline CPU scheduler, which is an OS-level control.

4 Design of Fray

In this section, we describe *Fray*, a new platform for concurrency testing of JVM programs designed to meet our specified objectives effectively (ref. Section 2 and Table 1).

Overall Architecture: Fray is designed to work on arbitrary JVM programs as a drop-in replacement for the `java` command or to extend existing JUnit tests with a simple annotation. Fray instruments the JVM bytecode of the target program on-the-fly and injects its own run-time library, which spawns a separate *scheduler thread*. Depending on provided input parameters, Fray either (a) executes the program many times using a provided search strategy (e.g., random, PCT [4], POS [75], and SURW [76]) and, if a bug is encountered, outputs a file containing the *thread schedule*, or (b) replays a single program execution with a given thread-schedule file for debugging.

Assumptions and Guarantees: Fray is designed to always be *sound* during testing; that is, every reported bug corresponds to an actual behavior that can manifest in the original program. Fray makes two important assumptions to provide a guarantee of *completeness*, which means that every concurrency bug can be reproduced by some thread schedule in its search space (though there is no guarantee that a particular search will find it), and to support faithful replay for debugging purposes. First, Fray assumes that the only source of non-determinism in the program is due to concurrency; that is, the program does not make use of `Random`, system I/O, or any timer (e.g., `Thread.sleep`)—this is reasonable for test suites, though Fray also supports relaxing these assumptions for real-world use cases (ref. Section 4.5). Second, Fray assumes that programs are free of *data races* (ref. Section 2.1 for why this matters). If the target program contains data races, Fray can still be used but its search space might exclude some theoretically observable behaviors (ref. Section 4.5 again for relaxing this assumption).

4.1 Concurrency Control in Fray

The key design principle in Fray is to orchestrate thread interleavings without replacing existing concurrency primitives with mocks, while also encoding the semantics of these primitives to faithfully express the set of all possible program behaviors. Based on this design philosophy, Fray implements a protocol called *shadow locking* which ensures that only one application-level thread executes at a given time and allows Fray to control the order in which threads interleave at synchronization points.

```
class Foo extends Thread {
  static Object o = new Object();
  static AtomicInteger a = AtomicInteger();
  static volatile int b;
  public void run() {
    // Let t = Thread.currentThread()
    $\mathcal{S}^t_{\text{run}}$.lock()

    $\mathcal{S}^t_{\text{atomic}(a)}$.lock()
    int x = a.getAndIncrement();
    $\mathcal{S}^t_{\text{atomic}(a)}$.unlock()

    $\mathcal{S}^t_{\text{monitor}(o)}$.lock() // monitorenter
    synchronized(o) {
      if (x == 0) {
        int k = $\mathcal{S}^t_{\text{monitor}(o)}$.getHoldCount();
        $\mathcal{S}^t_{\text{monitor}(o)}$.unlock() × k times
        do {
          o.wait();
        } while (!$\mathcal{S}^t_{\text{monitor}(o)}$.tryLock());
        $\mathcal{S}^t_{\text{monitor}(o)}$.lock() × k-1 times

      } else {
        o.notifyAll();
      }
    }
    $\mathcal{S}^t_{\text{monitor}(o)}$.unlock() // monitorexit

    $\mathcal{S}^t_{\text{volatile}(b)}$.lock()
    b = x;
    $\mathcal{S}^t_{\text{volatile}(b)}$.unlock()

    $\mathcal{S}^t_{\text{run}}$.unlock()
  }
  void main() { ... /* see Figure 1 */ ... }
}
```

Fig. 2. Fray's instrumentation (highlighted) of the program from Fig. 1, demonstrating *shadow locking* for concurrency control. Each $\mathcal{S}^t_*$ is a lock instantiated dynamically by Fray and initially held by Fray's scheduler thread; the shadow lock will be released by Fray when it wants to schedule thread t.

Shadow Locking: Fray instruments program classes to add extra synchronization around (but not to replace) concurrency primitives. Fig. 2 demonstrates how Fray would instrument class `Foo` from Fig. 1 with additional lock operations.

A *shadow lock* $\mathcal{S}^t_r$ is a lock associated with thread t and resource r. Whenever a thread t wants to access resource r, it must first acquire $\mathcal{S}^t_r$. Shadow locks can control access to the monitor of an object o, denoted as $\mathcal{S}^t_{\text{monitor}(o)}$; access to a volatile variable or atomic value v, denoted as $\mathcal{S}^t_{\text{volatile}(v)}$ or $\mathcal{S}^t_{\text{atomic}(v)}$ respectively; or the permission to start thread execution in the first place, denoted as $\mathcal{S}^t_{\text{run}}$ (for all application threads except the `main` thread).

Shadow locks are instantiated by Fray dynamically as needed, and immediately acquired by Fray's scheduler thread so that no application thread can acquire them by default. Fray tracks the application's shadow lock operations (such as `lock` and `unlock`) in order to maintain metadata about threads and their ownership of—or attempts to acquire—shadow locks. Fray maintains the following ***mutual exclusion invariant***: if any application thread t owns a shadow lock $\mathcal{S}^t_r$ for some resource r, then no other application thread t' can own the corresponding shadow lock $\mathcal{S}^{t'}_r$ for the same resource r.

Initially, only the main thread is running and it owns no shadow locks. If a thread is running, it will continue to run until it terminates or attempts to acquire a shadow lock (or executes

`Object.wait()`, details later). In the steady state, Fray maintains the invariant that at most one application thread can be running and all other application threads (if any) are blocked waiting to acquire a shadow lock (or be notified with a signal, details later). When no threads are running, Fray's scheduler picks a next thread t to run based on a specified ***thread schedule*** (i.e., sequence of threads $t_1, t_2, \ldots$ to interleave) and releases the shadow lock $\mathcal{S}_r^t$ that t is waiting to acquire. Fray uses its internal meta-data to ensure that it will only schedule a thread t if it can be guaranteed that t will make progress (i.e., it can acquire the resource r). If no such schedulable t exists, then a *deadlock* is reported. When an application thread releases a shadow lock, the Fray scheduler immediately re-acquires it.

To understand how Fray instruments Java programs to insert shadow locks, see Figure 2, which is the instrumented version of the example from Fig. 1, with Fray's changes highlighted. *Thread start:* Lines 7 and 33 are inserted to introduce the shadow lock $\mathcal{S}_{\text{run}}^t$ for controlling thread execution, so that new threads are blocked immediately after `Thread.start()` until Fray determines that no other thread is running. This is fine, because under the assumption of data-race-freedom, no event in the newly spawned thread can logically "happen before" any event in the creating thread until the latter has reached at least one synchronization point after the call to `Thread.start()`. *Monitor locks:* Lines 13 and 27 wrap a shadow lock $\mathcal{S}_{\text{monitor}(o)}^t$ around the `synchronized(o)` block (Line 14). A similar logic applies when a program uses `RentrantLock` or `Semaphore` classes. *Atomic and volatile:* The atomic increment at Line 10 and the volatile memory access at Line 30 both propagate information across threads [21, 55]. Fray conservatively treats these as synchronization points, inserting shadow locks $\mathcal{S}_{\text{atomic}(a)}^t$ (at Lines 9 and 11) and $\mathcal{S}_{\text{volatile}(b)}^t$ (at Lines 29 and 31) respectively around these accesses.

Handling `wait` and `notify` : In Java, the semantics of `wait/notify` make controlling concurrency non-trivial. `Object.wait()` can only be invoked by a thread holding a (possibly nested) lock on the object's monitor, and this call releases the lock (to full nesting depth) and blocks the thread. Correspondingly, `Object.notify()` can only be invoked by a thread holding a lock on the object's monitor, and when this lock is eventually released the JVM can resume any one thread that was blocked `wait`-ing on this object. Similarly, `Object.notifyAll()` marks all corresponding `wait`-ing threads as ready to be woken up by the JVM. The waiting thread(s) must have been in the middle of a (possibly nested) synchronized block; the lock is thus regained (to its original nesting depth) upon waking.

Concurrency control of `wait/notify` is hard because these methods are implemented natively in the JVM; from Java code, there is no way to control which thread(s) the JVM will wake up or to deterministically replay its choice later. Prior solutions either re-implement the entire JVM (e.g., JPF)—which introduces performance overhead and technical debt—or replace the call with custom mocks (e.g., Lincheck)—which is problematic, since the mocks cannot update the state of monitor locks in the same way (due to JVM restrictions on *block-structured locking*), and not doing so properly leads to compatibility issues when the JVM implicitly calls `wait/notify` on shared objects (e.g., when a `Thread` terminates).

Fray solves this problem using shadow locks as follows. Calls to `o.wait()` (e.g. Line 19 in Fig. 2) are (1) preceded by a full release of the shadow lock corresponding to `o`'s monitor (Lines 16–17), (2) wrapped by a loop (Lines 18–20) which only exits when the JVM wakes up the thread and Fray also makes the shadow lock available, as explained in the next paragraph. If the shadow lock is unavailable (i.e., `tryLock()` at Line 20 returns `false`), the thread loops back into the `wait()`. If available (i.e., `tryLock()` returns true), the loop exits, and the thread regains the shadow lock to its original nesting depth (Line 21). Fray also changes each invocation of `o.notify()` to `o.notifyAll()` (Line 24). This ensures that, instead of the JVM picking one arbitrary thread to resume, all waiting threads are woken, but only one thread t, deterministically chosen by Fray, continues execution (by making only its corresponding shadow lock available) while others return to a waiting state. These semantics are equivalent to the original program! The performance cost of the extra wake-ups is not significant enough to slow down Fray relative to other approaches (ref. Section 5.2).

So how does Fray decide when to release the shadow lock for a waiting thread to make it exit the do-while loop? When a thread t executes `o.wait()` in the original program, Fray adds t to a set of threads called $waiting_o$. Then, for an invocation of `o.notify()` in the original program, Fray deterministically removes one thread from $waiting_o$; similarly, for an invocation of `o.notifyAll()` in the original program, Fray removes all threads from $waiting_o$. At a scheduling step (that is, when no application threads are running), Fray can release the shadow lock of a thread only if it is not contained in any $waiting_o$ set and if it does not violate the mutual exclusion invariant described earlier. This scheme allows Fray to express not just immediate but also *delayed* wake-ups, where a notifying thread releases and re-acquires a monitor lock before the waiting thread can be awoken. Finally, Fray can also simulate *spurious* wake-ups (i.e., without a corresponding `notify`, which is legal) by encoding a special value in the thread schedule for removing a specific thread from $waiting_o$.

To summarize, while Fray explicitly encodes the semantics of `wait` and `notify` by maintaining state about waiting and notified threads, it does not replace the original calls—thereby allowing the application code and JVM to natively inter-operate with shared concurrency primitives such as the original object monitor locks, consistent with our design philosophy.

Other concurrency primitives: Fray supports all the standard locking primitives like `ReentrantLock`, `Semaphore`, etc. Fray handles `Condition.signal()/await()` exactly like `wait/notify`, and it handles `LockSupport.park/unpark` as well as `CountDownLatch` using similar meta-data but simpler instrumentation. The JDK implements `Thread.join()` using `Thread.wait()`, where the JVM calls `Thread.notifyAll()` upon thread termination; so, Fray does not need special handling. Fray does instrument calls to `Unsafe.compareAndSwap*`, just like atomic/volatile accesses, as well as `Thread.interrupt()`, whose details we omit here for brevity.

4.2 Soundness and Completeness

We provide informal sketches of our proofs of expressibility (Section 2.2, O2), whose details are available in an extended online version of this paper [37].

4.2.1 Soundness. We claim that every concurrency bug encountered by Fray can manifest in the original program. To see why, note that Fray only changes the original program by adding shadow-locking instrumentation as shown in Figure 2. Adding extra synchronization does not change program semantics, though it can introduce new deadlocks. Fray's execution deadlocks only if the original program could also deadlock, and this follows from how Fray couples each shadow lock $\mathcal{S}_r^t$ to the program's attempts to acquire the underlying resource r. Apart from synchronization, Fray also inserts a `do-while` loop for `Object.wait()` and changes `Object.notify()` to `notifyAll()`, but this is equivalent to the original semantics.

4.2.2 Completeness. We have formally modeled a subset of Java operational semantics to prove Fray's completeness guarantee [37]. Using these semantics, sequentially consistent executions of the original program can be represented as *traces*, which are sequences $s_0 \xrightarrow{t_0,e_0} s_1 \xrightarrow{t_1,e_1} s_2 \ldots$, where each s_i is a program state, and each $\langle t_i, e_i \rangle$ represents a state transition to s_{i+1} by executing an instruction e_i from thread t_i. An instruction can either be a synchronization instruction (e.g., `monitorenter` or `atomic`) or a thread-local instruction (e.g., `read` or `branch`), depending on whether or not it creates a happens-before relationship as per the Java memory model [21]. Let $next(s, t)$ be the instruction that thread t would execute if scheduled at state s, or be undef if the thread t cannot run. Then, we define a *sync-point-scheduled* (SPS) trace which is a special kind of trace where $t_i \neq t_{i+1} \Rightarrow next(s, t_i) \in \{\text{undef}, \text{monitorenter}, \text{wait}, \text{atomic}, \dots\}$; that is, thread t_i yields control to another thread only if it is blocked/terminated or about to execute a synchronization instruction. Fray's shadow locking protocol produces SPS traces. We can then state the completeness theorem:

Theorem 1. *For every assertion violation (evaluated using a predicate made up of thread-local instructions) or deadlock that manifests in a trace π, there exists an SPS trace π' that exhibits the corresponding assertion violation or deadlock.*

Proof. (Sketch) Consider any program trace $s_0 \xrightarrow{t_0,e_0} s_1 \xrightarrow{t_1,e_1} s_2 \dots s_n$. By the assumption that programs are data-race-free, we can swap any pair of consecutive transitions $s_i \xrightarrow{t_i,e_i} s_{i+1} \xrightarrow{t_{i+1},e_{i+1}} s_{i+2}$, where $t_i \neq t_{i+1}$ and e_i, e_{i+1} are not synchronization instructions, to get a new valid subtrace $s_i \xrightarrow{t_{i+1},e_{i+1}} s'_{i+1} \xrightarrow{t_i,e_i} s''_{i+2}$ such that $s_{i+2} = s''_{i+2}$. In other words, thread-local instructions across different threads are independent and can be reordered without affecting the resultant state. We can thus perform a series of such reorderings on a program trace exhibiting a concurrency bug until we get (a prefix that is) an SPS trace exhibiting the same bug. □

Finally, we can show that Fray's shadow locking protocol can enumerate *all* SPS traces, since Fray is allowed to release a shadow lock for any thread that is able to acquire the underlying resource. Thus, Fray's search space is *complete* under the data-race-free assumption.

4.3 Scheduling and Search Strategies

Once we achieve full concurrency control—that is, we can choose arbitrary *schedules* to deterministically execute specific thread interleavings—we can now hunt for concurrency bugs by running *search strategies*. In Fray, search strategies are plug-ins that implement the logic for choosing the next schedule to run, given some prior search state. The schedules are instantiated dynamically; that is, the program executes a single thread until it is blocked (e.g., when attempting to acquire a shadow lock). Then, Fray makes a callback to the search strategy, providing the set of *enabled* threads (that is, threads which Fray knows will make progress if scheduled). The search strategy then returns the next thread to execute.

The search strategies can either be systematic (e.g., depth-first search) or randomized. We do not invent any new algorithms; rather, we look to prior work for picking good strategies. Thomson et al. [66] empirically studied several concurrency testing algorithms including systematic depth-first search, iterative context bounding [44], iterative delay bounding [11], probabilistic concurrency testing (PCT) [4], and the coverage-guided Maple algorithm [73]. They found that PCT was most effective, followed by a naive random walk. Subsequently, Yuan et al. [75] introduced the partial-order sampling (POS) algorithm that was shown to outperform PCT. Finally, Zhao et al. [76] very recently introduced the selectively uniform random walk (SURW) technique, which provides strong guarantees on uniformity of sampling over the interleaving space.

So, we currently support the following strategies in Fray: (1) **Random Walk**—At each scheduling step (i.e., callback), pick one of the enabled threads uniformly at random. (2) **PCT**—randomize priorities for all threads, and, at each step, schedule the thread t with the highest priority; if this is one of d randomly chosen steps (where d is a user-defined *depth* parameter), then set t to the lowest priority. (3) **POS**—randomize priorities for all threads, and, at each step, schedule the thread t with the highest priority; also reassign random priorities for all other threads competing for the same

```java
class FooTest {
  @ConcurrencyTest(/* optional args */)
  public void testTwoFooThreads() {
    ... /* same code as Foo.main() in Figure 1 */ ...
  }
}
// Optional args for @ConcurrencyTest
//   iterations=<integer> (default is 1000)
//   scheduler=[Random|PCT|POS|SURW]Scheduler.class (default is POS)
//   replay=<filename> to enable replay mode (default is empty, for testing)
```

Fig. 3. Typical usage: A code example showing how a user can annotate a JUnit 5 test method to run existing test code with Fray performing either testing with an optionally specified scheduler for a number of iterations, or performing replay with a given interleaving file saved by a previous run of Fray.

```
Bug found in testTwoFooThreads() iteration 3/1000, you may find detailed report
↪  and replay files in /path/to/project/fray-report
Exception java.lang.AssertionError: expected:<1> but was:<0>
        at FooTest.testTwoFooThreads(FooTest.java)
```

Fig. 4. Example of console output when Fray detects a concurrency bug (after running the test in Fig. 3), showing the assertion failure and indicating where detailed reports and replay files can be found for debugging.

resource that t is about to acquire. (4) **SURW**—takes a list of interesting events as input and, at each step, determines a thread to execute using weights proportional to the number of remaining interesting events per thread.

Fray is designed to be easily extensible with new search algorithms. For example, we encountered the SURW paper (which was recently presented at ASPLOS'25) much after Fray's initial implementation and evaluation; it took one of the authors only one day and ~200 LoC to implement this algorithm as one of Fray's scheduling strategies.

4.4 End-to-End Usage

Fray can either be run on the command-line or as an extension to JUnit 5 tests with Gradle or Maven plugins that we make available on the Maven Central Repository.

Command-Line Mode: To use Fray to test the program in Figure 1, simply replace `java Foo` with `fray Foo`. This will repeatedly run the program using a random-walk thread scheduler until it is terminated or a bug is found. Command-line options `--scheduler` and `--iter` can specify the scheduling algorithm and the number of iterations respectively.

When a test fails, Fray provides detailed failure information, including the specific exception encountered and the location where detailed reports and replay files are saved. The user can then run `fray --replay=<file> Foo` to perform replay debugging. The replay file records all the random decisions made during the original run, including which thread to execute at each step and when to trigger spurious wake-ups. While the recording file is not directly human-readable, we provide an IntelliJ IDEA plugin for visualizing thread interleavings within the IDE [36].

Testing Framework: More commonly, we expect users to integrate Fray into their existing test suites that provide testable entry points and which already configured for complex build scenarios. Figure 3 demonstrates how a developer can test the `Foo` class shown in Figure 1 using the Fray JUnit 5 extension. To execute a test using Fray, developers need only annotate the test method with `@ConcurrencyTest(...)` and optionally specify configuration parameters such as the number of iterations to execute and the scheduling algorithm to employ (Lines 8 and 9, which also indicate the default choices). When a test fails, Fray provides detailed failure information along with a recording file, as shown in Figure 4. For debugging, users can specify the recording file name as an option to the `@ConcurrencyTest()` annotation (see Fig. 3 Line 10). As such, a test method annotated with `replay` is like a unit test that executes a single specified thread schedule. This is useful for step-through debugging within an IDE, as the replay configuration is only intended for local debugging and not for commiting to version control.

4.5 Relaxing Assumptions

Data-Race-Freedom: Since Fray assumes that programs are data-race-free by default, performing context switches (via shadow locks) only at synchronization points is sufficient to guarantee completeness of its search space (ref. Section 4.2). If a program has data races, then Fray can still be used but its search space might miss some bugs that only manifest when (1) threads interleave between unsynchronized shared-memory accesses or (2) the application exhibits non-sequentially-consistent behavior. Although Fray cannot solve the latter issue, there is some support to address the former: a special flag (`--memory`) can be used to force Fray to insert shadow locks (and therefore perform context-switching) around *every* shared-memory access, not just the atomic and volatile fields. Naturally, this introduces a large performance cost. We do not explicitly evaluate this mode in the rest of this paper as it is not the primary use case; we believe users should use dedicated data-race detectors that can also address Java's weak memory model before running Fray.

Dealing with non-determinism apart from concurrency: Even if Fray is run on generally testable code, there are some forms of non-determinism that are common in real-world use cases. Fray handles the most common code patterns that we have observed in open-source projects.

First, test code sometimes uses `Thread.sleep()` and applications often use timer-based synchronization primitives, e.g., `Object.wait(timeout)`, `Condition.awaitNanos()`, or `LockSupport.parkNanos()`. Fray automatically rewrites the sleep to a *no-op* and the timed waits to basic `wait()`/`await()`/`park()` without timeouts; none of this affects completeness. For the latter group, since Fray models "spurious" wake-ups that can happen at any time, this covers the behavior where the timeout triggers, given that the two cases are semantically indistinguishable [53].

Second, application code that makes use of `Object.hashCode()` can have subtly different behavior across runs (e.g., whether or not two keys in a hash-map collide in the same bucket). To enhance deterministic replay, Fray's recording files save object hash codes in the order in which the application read them and then reproduces these values during the replay phase. Similarly, Fray records and replays system clock values (e.g., `System.nanoTime()`).

Third, it is common for classes to run static initializers to set up global state. Because this logic only runs once, if there is any concurrency in the static intializers then the saved recordings for schedules encountered during the *n*-th iteration of random testing will not align with the order of events during replay-from-scratch. We provide a special flag (`--dummyRun`) in replay mode to run a warm-up iteration before attempting deterministic replay hoping to skip the static initializers.

Finally, Fray does not record/replay calls to `java.util.Random` or file I/O, since most test code uses these APIs in a manner that does not affect control flow (e.g., generating UUIDs, reading static config files, or logging messages). So, Fray's replay mode simply performs the original calls for all such APIs, at the risk of introducing flakiness if the application deviates from these norms.

4.6 Maintainability

Fray currently runs on Java 21 (the latest long-term support version at the time of writing) and is compatible with targets compiled for older Java versions as well. Unlike some tools such as

CalFuzzer or JMVx, Fray does not depend on `Unsafe` APIs or JVM-vendor-specific features, making it robust to changes in JVM implementations. With Fray's design, we only need to insert shadow locks around the usages of concurrency APIs listed in the language reference [22] and the JDK documentation [54]. For example, we recently added Java 23 support with ~50 lines of code. This was necessary because JDK 23 replaced `LockSupport.park/unpark` with `Unsafe.park/unpark` for thread parking operations in certain concurrency primitives, requiring us to update our instrumentation accordingly.

5 Evaluation

Our evaluation aims to compare Fray against a representative concurrency testing tool from each design choice category shown in Table 1. Among the available tools, only RR (*OS-level interception*), JPF (*VM Hacking*), and Lincheck (*concurrency mocking*) are actively maintained and compatible with contemporary versions of Java (11–21). Lincheck, however, is primarily designed for testing data-structure linearazibility instead of arbitrary Java applications; whereas RR and JPF are explicitly general-purpose. Unfortunately, all of the *directed yielding* tools CalFuzzer, IMUnit, and Weaver support only Java 6, which is deprecated since 2018 and not supported by any of our evaluation benchmarks. To confirm our claims from Section 3, we were able to run CalFuzzer on a virtual machine with Java 6 and verified its lack of expressibility in controlling the thread wake-up order after `notify()`, as well as the inability to simulate certain interleavings such as delayed and spurious wake-ups.

Our evaluation addresses the following research questions:
RQ1: *How does Fray compare to RR and JPF concurrency control in terms of bug-finding effectiveness on independent benchmarks?*
RQ2: *How does Fray compare to RR and JPF in terms of run-time performance on the same benchmarks?*
RQ3: *Can Fray be effectively applied to real-world software in a push-button fashion? How does it compare to other tools?*
RQ4: *Can Fray be used for special-purpose testing of concurrent data structures? How does it compare to Lincheck in terms of bug-finding scope?*

Experimental Setup: For RQ1 and RQ2 only—to ensure that we are only comparing different design choices for concurrency control (ref. Table 1) and not the variations in search strategies, we fix all tools to use a *random* search: "Fray-Random" and "JPF-Random" explicitly use a random walk to schedule threads, whereas "RR-Chaos" uses RR's *chaos* mode [46] to randomize thread priorities during record-and-replay. For RQ3 and RQ4, we also run modern search strategies (ref. Section 4.3): Fray-PCT3 (with depth d=3)[2], Fray-POS, and Fray-SURW[3]. All experiments are performed on an Intel Xeon processor with 16 cores and 187 GB of memory.

Benchmarks and Metrics: For RQ1 and RQ2, we target concurrency-bug benchmarks from prior work: SCTBench [66] and JaConTeBe [39]. SCTBench is a set of micro-benchmarks of concurrency bugs using the `pthreads` library and with explicit assertions. For our evaluation, we manually translated the subset of 28 programs with the "CS" prefix to Java (as these were self-contained and also used in prior work [7, 8, 70, 71])—the translation converted `pthreads` mutex lock operations to Java `synchronized` blocks. For programs with data races, we replaced unsynchronized memory accesses with corresponding `volatile` accesses in the Java translations; this is sound because the bugs intended by the benchmark are other types of race conditions (e.g., atomicity violations) not the data races themselves. So when we mark variables as volatile—as a real user

[2]We choose depth 3 because prior work [66] showed it to be the most effective for revealing bugs.
[3]We randomly sample 20 memory locations and mark all their access events as interesting, as suggested by the original authors.

would do if they found the data race via a race detector and then patched it—the atomicity bug still remains. JaConTeBe is a benchmark suite derived from real-world concurrency bugs in open-source Java software [39]. The original suite includes both application bugs and OpenJDK bugs—we use a subset of 25 bugs whose test cases have oracles and which come from application projects (such as Apache Log4J, DBCP, Derby, and Groovy). For each of the 53 benchmark programs, we run each tool's random search for up to 10 minutes and repeat this experiment 20 times to get statistical confidence.

For RQ1, we measure the wall clock time and the number of executions to find each bug if discovered in some run.

For RQ2, we measure run-time performance by comparing the number of executions per second on the 53 programs with the randomized search running the whole 10 minutes whether or not a bug is encountered,[4] and then count the number of iterations of the search completed. This approach allows us to reliably measure steady-state performance for all tools.

For RQ3, we target over 2,600 test cases from real-world open-source software projects: Apache Kafka@43676f7 ("streams" module, 1M LOC), Apache Lucene@33a4c1d (857K LOC), and Google Guava@635571e (353K LOC).

For RQ4, we evaluate Fray on the 9 new bugs discovered by Lincheck as reported in its original paper [33].

5.1 RQ1: Bug-Finding Effectiveness

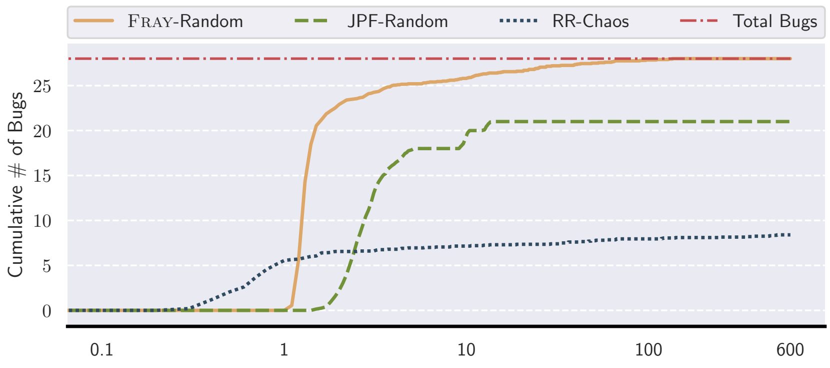
Figure 5 shows the cumulative distribution of bug-discovery times using the three tools across 53 buggy programs from the two benchmarks, averaging over 20 repetitions.

In SCTBench (Fig. 5a), Fray-Random is most effective, detecting all 28 bugs after 100 seconds on average. JPF-Random quickly detects 20 bugs within the first 10 seconds on average but cannot identify the remaining 8 bugs even after running for 10 minutes on each target. RR-Chaos is the least effective, finding only 8 bugs during the experiment. JPF missed one bug due to its incomplete implementation of `AtomicLong.set()`. The remaining 6 bugs involve atomicity violations—concurrent read/write operations requiring specific thread interleavings to manifest. JPF likely failed to detect these due to limited exploration caused by its performance overhead (see also RQ2). For example, in one benchmark program (`TwoStage100Bad`) Fray explored 14,800 schedules in the time bound whereas JPF could explore only 1,700 schedules.

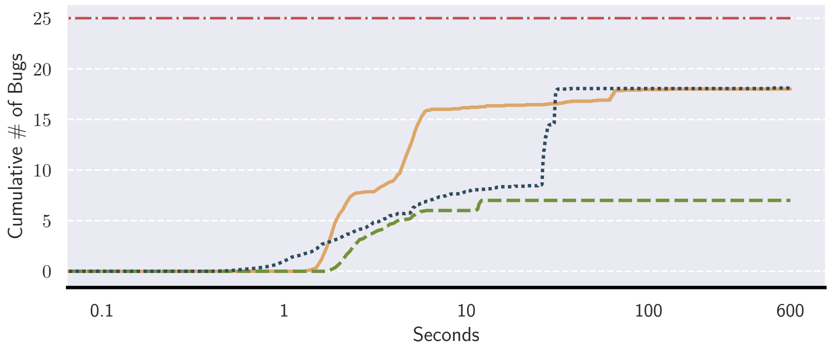
In JaConTeBe (Fig. 5b), Fray-Random and RR-Chaos show the highest effectiveness, detecting 18 bugs on average. JPF-Random only identifies 7 bugs. It is worth noting that JPF failed to run 16 tests due to compatibility issues often stemming from missing support for certain JDK APIs, highlighting the importance of *applicability*. A burst occurs around 30 seconds for RR because it lacks a built-in deadlock detector, relying instead on the JaConTeBe harness, which runs its deadlock detector every 30 seconds. Why did Fray miss 7 bugs? We identified that 6 bugs are caused by data races (which is out of scope for Fray); of these, JPF identified one and RR-Chaos identified two. The seventh miss for Fray was a real false negative—an undiscovered deadlock—that JPF also failed to detect but was successfully identified by RR.

We have claimed that Fray's design optimizes the search space when compared to OS-level concurrency control, since we only focus on application threads. Figure 6 evaluates this via a proxy: measuring the average number of executions for the random search to find each bug. As predicted, across both benchmarks, Fray and JPF require a similar number of executions to find most bugs. In contrast, RR-Chaos can sometimes require up to 100× more executions to detect the same bugs.

[4]If we stop the search when a bug is found, then the performance appears to be dominated by the JVM startup time when bugs are found quickly.



(a) SCTBench



(b) JaConTeBe

Fig. 5. Comparison of bug-finding effectiveness over time (higher and quicker is better) on 53 programs across two benchmark suites, with a search timeout of 10 minutes per target. The y-axis shows the cumulative number of unique bugs found, while the x-axis shows execution time on a logarithmic scale from 0.1 to 600 seconds. Lines and shaded areas represent means and 95% confidence intervals, respectively, across 20 repetitions. The "Total Bugs" line (red dash-dot) represents the theoretical maximum discoverable bugs.

Note that JaConTeBe is designed to be a reproducible bug benchmark; so, some bugs can be found in just 1 iteration.

Overall, Fray consistently demonstrates superior bug-finding effectiveness on independent benchmarks, identifying 70% more bugs than JPF and 77% more bugs than rr. JPF is effective when analyzing micro-benchmarks from SCTBench, but suffers from low applicability when analyzing benchmarks derived from real-world software in JaConTeBe.

Fray demonstrates superior bug-finding effectiveness, identifying 46 out of 53 concurrency bugs (87%) across both benchmarks, while JPF detected only 27 bugs (51%) and rr found 26 bugs (49%).

5.2 RQ2: Run-time Performance

Figure 7 shows the average execution speed for each tool on the 53 benchmark programs. Fray is the most efficient, with a median speedup of 10× over JPF and 457× over rr. The performance overhead of JPF is easy to understand: it runs a custom JVM interpreter and so cannot take advantage of

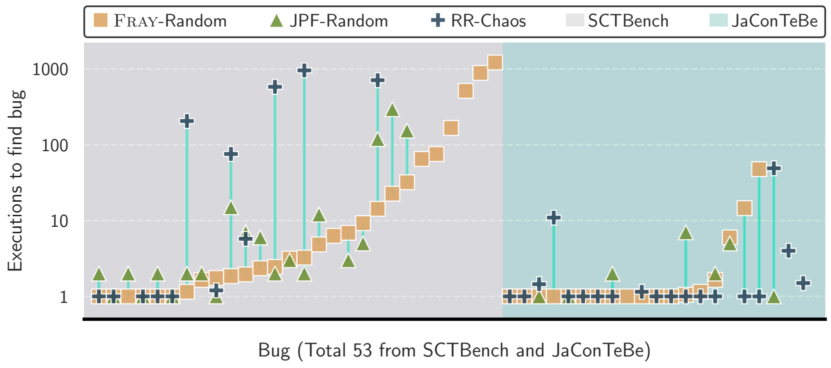


Fig. 6. A proxy for search space evaluation: Comparison of number of executions (y-axis, in log scale, lower is better) needed by the three tools to find each of 53 bugs (x-axis), averaged across 20 repetitions. Missing markers imply the tool failed to find the bug within the 10-minute timeout in any repetition. The background shading distinguishes between the two benchmark suites.

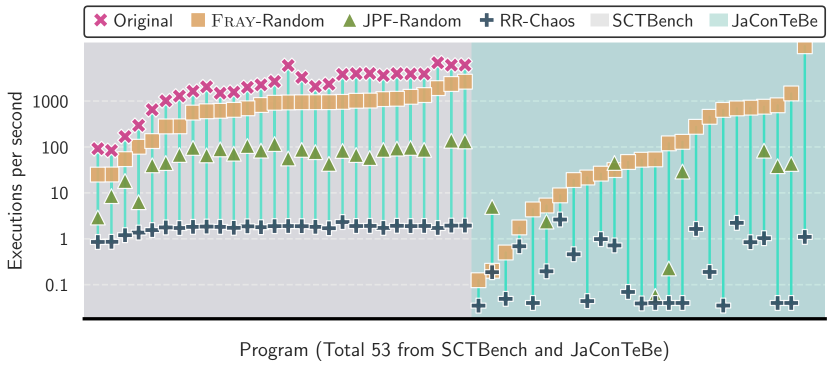


Fig. 7. Run-time performance of each tool, measured as average executions per second (y-axis, in log scale, higher is better) running the random search for 10 minutes on each of the 53 benchmark programs (x-axis). Missing markers indicate instances where the tool failed to run the program at all. The background shading distinguishes between the two benchmark suites.

JIT optimizations. But what about rr, which runs the HotSpot JVM natively? First, rr cannot distinguish the application from the managed runtime, and so it must restart the JVM for each execution. Second, the OS-level concurrency control over the whole JVM process requires recording lots of unnecessary information (e.g., all the non-determinism in the JVM), which is expensive. Fray's use of shadow locks for concurrency control makes it very efficient.

In order to measure the overhead over vanilla uninstrumented (and therefore non-deterministic) execution, we also include a baseline called *Original*, which simply runs the target test repeatedly in a loop for 10 minutes on a single CPU core. However, to do this fairly, we also need to keep the loop going even when we encounter deadlocks—we were only able to do this for SCTBench, by modifying the source code to prevent the actual deadlock, but not JaConTeBe, which is not open-source. The results in Fig. 7 show that in SCTBench, the instrumentation and scheduling

Table 2. Results of concurrency testing on real-world software. "Tests run" represents the number of test cases that can be executed and whose thread interleavings searched over without internal tool errors. The second column shows the number of test failures identified by each technique (and the subset of failures that are caused by wall-clock-based timed waits). The last column presents the total number of reported bugs (along with the subset confirmed by the developers).

	Technique	Test Run (10 mins each)	Test Failures (Time-related)	Reported Bugs (Confirmed)
Kafka-Stream	Fray-PCT3	279	126 (60)	6 (6)
	Fray-POS	279	162 (66)	10 (9)
	Fray-SURW	279	165 (90)	8 (8)
	Fray-Random	279	92 (10)	8 (8)
	JPF-Random	0	0 (0)	0 (0)
	rr-Chaos	278	4 (1)	2 (2)
Lucene	Fray-PCT3	1186	3 (3)	0 (0)
	Fray-POS	1186	7 (3)	4 (3)
	Fray-SURW	1186	4 (4)	0 (0)
	Fray-Random	1186	3 (3)	0 (0)
	JPF-Random	0	0 (0)	0 (0)
	rr-Chaos	1179	0 (0)	0 (0)
Guava	Fray-PCT3	1199	194 (135)	3 (3)
	Fray-POS	1199	194 (135)	3 (3)
	Fray-SURW	1199	199 (137)	3 (3)
	Fray-Random	1199	196 (137)	3 (3)
	JPF-Random	0	0 (0)	0 (0)
	rr-Chaos	1191	1 (1)	0 (0)

overhead introduced by Fray resulted in a median slowdown of only 3.2× compared to the original, in contrast to 31.5× for JPF.

Fray demonstrates superior runtime performance with a median speedup of 10× over JPF and 457× over rr, making it the most efficient among the compared tools for concurrency testing.

5.3 RQ3: Applicability to Real-World Software

We (attempt to) run all tools, including all search strategies of Fray, on *every* unit test that spawns more than one thread from the suites of Kafka-streams [16], Lucene [15], and Guava [19]—a total of 2,664 distinct test cases (each running a 10-minute randomized search over schedules). To the best of our knowledge, this is the *largest evaluation of controlled concurrency testing on real-world software.*

Table 2 shows the results of these experiments. The table is divided into sections representing each of the target projects: Kafka-Streams, Lucene, and Guava. The column "Tests Run" indicates the number of tests that a given tool could run at all in a push-button fashion, without crashing with an internal error. Note that the tests are *expected* to pass as they have been regularly running in the projects' corresponding CI pipelines. The next column counts the subset of test runs in which the search strategy found a concurrency bug; that is, the test can be shown to fail (by an assertion violation, run-time exception, or deadlock) due to a race condition. All such failures can be deterministically reproduced with a fixed schedule. A subset of these failures depends on the timers used by the tests (most commonly, `Object.wait(timeout)`)—we identify these explicitly since some of the failures may be highly improbable in practice. Failing tests have to be analyzed manually to deduplicate underlying bugs. The last column shows the number of unique bugs reported to (and confirmed by) the developers.

The results show that Fray excels in both applicability and effectiveness. First, Fray can run all the real-world software tests in a push-button fashion. Second, even Fray-Random identifies interleavings that cause failure in 291 tests (vs. only 5 by rr-Chaos). However, both Fray-POS and Fray-SURW work better, finding failures in 360+ tests each—requiring an average of 190 and 82 iterations respectively to search for them. JPF cannot run *any* of the 2,664 tests; it always fails with internal errors due to unimplemented native methods or unsupported JDK features.

We manually deduplicated the bugs identified by the Fray and reported 18 distinct bugs to project maintainers. For each bug, we have reported detailed instructions for reproduction. Developers have confirmed 16 bugs at the time of writing (and 12 have been fixed); the rest are awaiting triaging. Among the 18 bugs we reported, six are caused by atomicity violations, five by order violations [42], one by unhandled spurious wakeups, five by thread leaks, and the remaining one is yet unclassified. Note that some of the tests in which we observed bugs were encountered as flaky failing tests in past issues, but developers had not identified a way to reproduce and determine the root cause. Fray unlocks the capability to reliably identify and debug such failures.

In the evaluation of controlled concurrency testing on real-world software involving 2,664 distinct test cases, Fray demonstrates superior applicability and effectiveness by successfully running all real-world tests in a push-button fashion and identifying 360+ test failures using either POS or SURW, while other approaches either found significantly fewer failures (e.g., rr-Chaos found only 5) or failed to run any tests at all (JPF).

5.4 RQ4: Linearizability of Data Structures

Lincheck [33] is a state-of-the-art specialized concurrency testing framework focusing on *linearazibility* [23] of concurrency data structures (e.g., `ConcurrentHashMap`), determining whether a set of concurrent method calls have the same effect as some series of sequential method calls. For this purpose, Lincheck provides a lightweight interface for declaring data-structure APIs and automatically converts this spec to a test runner that includes the sequence of API calls and the test oracle. While the previous RQs evaluated Fray's capabilities for general-purpose concurrency testing, we would like to (1) evaluate whether Fray can also be used for testing of concurrent data structures, and (2) conceptually compare the bug-finding scope of Fray and Lincheck.

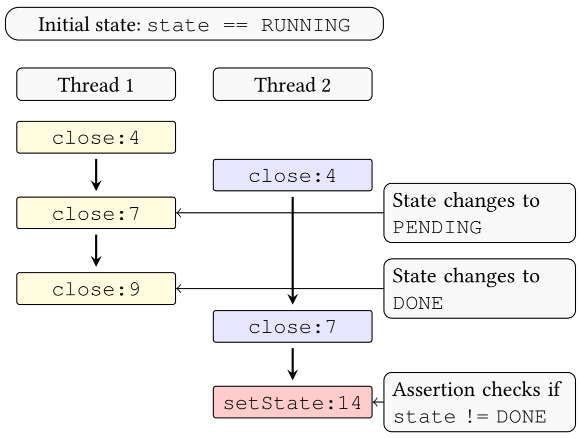
We evaluate Fray on the 9 concurrent data structures in which the original Lincheck paper [33] reports new bugs. For each of these data structures, Lincheck reports the sequence of API calls that results in the linearizeability bug. To allow Fray, JPF, and rr to also attempt to find these bugs, we manually translated each of these sequences of API calls into test drivers and added assertions that the results confirm to that of a linearizable execution.

Using the same 10-minute search as before, Fray-POS finds 8 out of 9 bugs, compared to Fray-PCT3 (7 bugs), Fray-SURW (6 bugs), Fray-Random (6 bugs), JPF-Random (2 bugs), and rr-Chaos (2 bugs). The only bug Fray-POS misses is a liveness error in Kotlin's `Mutex`: Kotlin implements *coroutines* by interleaving multiple logical tasks on the same user-space thread, and context switching at blocking I/O operations. For Fray, this program appears single-threaded so it cannot explore any interleavings.

While Fray successfully found interleavings that trigger 8 out of 9 linearizability bugs, it required manually constructed test drivers to do so. This highlights the complementary nature of these tools: Lincheck offers a powerful, specialized approach for concurrent data structure testing, whereas Fray provides robust general-purpose concurrency testing capabilities. Fray's broader scope enables

```
 1 class KafkaStream {
 2   volatile State state;
 3   boolean close() {
 4     if (state == DONE) {
 5         return true;
 6     }
 7     setState(PENDING);
 8     //...
 9     setState(DONE);
10   }
11   synchronized void
12   setState(State newState) {
13     if (newState == PENDING) {
14       assert(state != DONE);
15       state = newState;
16     } else if (...) {}
17 } }
```

(a) Simplified KafkaStream implementation that throws AssertionError when two threads try to close the stream concurrently.



(b) Execution trace demonstrating the race condition: Thread 1 successfully transitions the state from RUNNING through PENDING to DONE, while Thread 2 attempts to set the state to PENDING after it is already DONE, triggering the assertion failure in the setState method.

Fig. 8. Race Condition in Kafka Stream Close Operation

it to effectively test complex concurrent applications that create and manage their own threads, such as Kafka and Lucene, which falls outside Lincheck's original scope.

Fray is capable of finding interleavings that result in linearizability bugs when provided the test driver. We believe that Fray and Lincheck serve complementary purposes in the concurrency testing ecosystem–Fray excels at general-purpose concurrency testing of existing developer-written tests, while Lincheck specializes in end-to-end testing of concurrent data structures.

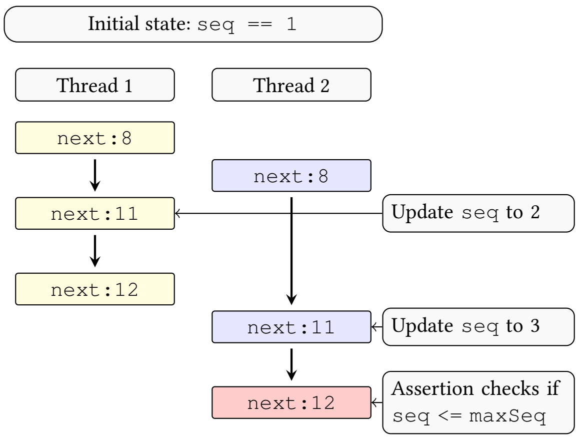
6 Case Studies

KAFKA-17379: Figure 8a shows the simplified implementation of `KafkaStream` and its `close method`. The `KafkaStream` is designed as a state machine and transitions through different states during its lifecycle. A concurrency bug arises when two threads attempt to close the same stream. As shown in the Figure 8b, when Thread 1 successfully changes the state from RUNNING to PENDING and then to DONE, Thread 2 can still attempt to set the state to PENDING. This violates the assertion in the `setState` method that checks whether the state is not already DONE when transitioning to PENDING. The bug manifests as an `AssertionError` when Thread 2 reaches this assertion check. This race condition occurs because the `close` method does not acquire proper synchronization before checking and modifying the state.

This bug, introduced approximately four years ago, remained hidden until Fray detected it. Notably, the developers had already implemented tests capable of triggering the bug, but without any mechanism for controlled concurrency testing the buggy interleaving was never identified. The bug turned out to be tricky to fix. After our report, the developers iterated on fix strategies for over a week (over 26 comments and replies in the GitHub issue) and ultimately settled on redesigning the existing state transition model. Moreover, we were able to help the developers confirm that the fix did not introduce new concurrency bugs by re-running the patched implementation with Fray.

```java
class Queue {
  volatile long maxSeq = 2;
  volatile long seq = 1;
  Queue advance(int ops) {
    maxSeq = seq + ops;
  }
  long next() {
    if (seq + 1 > maxSeq) {
        advance();
    }
    long current = ++seq;
    assert seq <= maxSeq;
    return current;
  }
}
```

(a) Simplified DocumentsWriter-DeleteQueue implementation that throws AssertionError when two threads try to acquire a sequence number concurrently.



(b) Execution trace showing the race condition between two threads. The race occurs when both threads try to get a sequence number from the same queue, leading to an assertion failure when the sequence number exceeds the maximum allowed value.

Fig. 9. Race Condition in Lucene Document Writer.

It is worth noting that in our experiments, this bug was only detected (within the 10-minute time bound) using the POS scheduling algorithm, highlighting that the advances in concurrency testing algorithms indeed benefit practical bug detection in complex concurrent applications. Despite the theoretical existence of such algorithms for years, developers lacked practical testing frameworks to apply them effectively in real-world scenarios.

LUCENE-13571: Figure 9a shows the simplified implementation of DocumentsWriterDelete-Queue, which demonstrates an atomicity violation bug [43] that caused an assertion failure in Apache Lucene. The bug occurs when multiple threads access the queue concurrently. As shown in the execution trace, Thread 1 and Thread 2 both attempt to increment sequence numbers, but without proper synchronization between queue advancement and sequence number generation. When the queue's maximum sequence number is set by one thread, another thread may continue incrementing the sequence counter beyond this maximum, triggering the assertion error.

Although the race condition appears evident in the simplified code, it manifests with considerably more subtlety in the actual implementation due to complex interactions between queue deletion and reuse. A flaky test failure was initially documented in February 2024; however, it remained unresolved as developers encountered significant challenges in reproducing the issue and could not isolate the root cause. Despite implementing several proposed fixes, the issue persisted. Fray rediscovered the test failure while running concurrency tests in the Lucene code base and provided developers with a deterministic execution trace. With the trace, developers successfully identified the root cause of the failure and implemented an effective fix, resolving an issue that had persisted for six months since its initial discovery.

After the issue was fixed, we received an email from the developers saing: "With all the hype around LLMs, etc. it's refreshing to see practical and cutting edge research in something as useful and powerful as debugging concurrent programs on the JVM." Subsequently, Elastic Search Labs also invited us to give a talk about our work and published a blog post [34] about how Fray helped find the Lucene bug.

7 Discussion

7.1 Threats to Validity

In our evaluation, one threat to *construct validity* arises from our porting of SCTBench to Java. We mitigated this threat by only targeting a subset of SCTBench which was standalone and could be validated manually. Our threat to *internal validity* stems mainly from the fact that RR and JPF were originally designed for different use cases (record-and-replay and model checking respectively); so, even though they support random testing they might not be engineering to do that optimally. We mitigate this threat by relying not only on time-to-bug-discovery, but also evaluating search space (Fig. 6, where JPF is quite comparable to Fray) and applicability (Table 2, where the inability to run random testing also implies the inability to run JPF's model checking or RR's replay debugging). Our evaluation naturally has a threat of *external validity*—while we demonstrated Fray's effectiveness on a number of benchmarks and real-world targets, we cannot make scientific claims about Fray's superiority on every target in general.

7.2 Limitations

As evidenced by our evaluation, Fray can miss bugs in programs with data races, and cannot interleave single-threaded co-routines. Additionally, Fray's handling of timed waits and sleep statements introduces practically improbable schedules. For example, consider a thread that sleeps for 10 seconds, waiting for a flag to be updated by another thread. According to the JLS [21], "neither `Thread.sleep` nor `Thread.yield` have any synchronization semantics"; thus, a schedule where the sleep completes before the other thread has made any progress is valid. However, we have observed some developers being unwilling to fix such issues given their improbability. In future work, we plan to use statistical methods or allow user configurations to suppress such warnings.

7.3 Insights on "Applicability"

While past work on concurrency testing has focused mainly on optimizing performance and search strategies, our research highlights the importance of focusing system design on "applicability", which hitherto has been overlooked as simply an implementation detail. To find real bugs, you have to be able to run a tool on real software, and minimize developer effort to manually adapt their software for the testing tool. To do this, the tools have to be able to support arbitrary program features. Even supporting say 90% of language features does not mean that you can test 90% of real-world targets; the inability of JPF to run on most targets in our study (ref. Table 2) is evidence of this–any mishandled feature can cause the application or JVM to crash and the test to be meaningless.

We observed that the limits to applicability inherent in tools like JPF and JMVx stem from the need to support various interactions between applications and the JVM through shared concurrency primitives (ref. Section 3). Due to the complex web of inter-dependencies between Java features such as object monitors, wait/notify, thread creation/termination, atomics, unsafe, etc. the decision to replace any concurrency primitive opens up a Pandora's box of special cases to handle, inevitably leaving loose ends in the limit.

In contrast, Fray only requires identifying a set of core concurrency primitives (defined in the language manual [21]) to wrap around with shadow locks. By not having to re-implement any concurrency primitives, the application can fully inter-operate with the JVM, whose code we do not need to control. Our evaluation shows that Fray can therefore be applied to real-world software in a *push-button* fashion.

7.4 Reflection on Applying a State-of-the-Art Search Algorithm to Real-World Targets

Our original implementation of Fray supported random walk, PCT, and POS as scheduling strategies. When we encountered the SURW paper [76], which was recently presented at ASPLOS'25, we were happy to find that implementing the algorithm in Fray was relatively straightforward, requiring only 8 human hours and ~200 lines of code. To validate our implementation of SURW, we manually translated the examples illustrated in the original paper+artifact and compared the standard deviation of schedules sampled by both implementations, confirming their similarity. Following the original authors' suggestion, we randomly sampled a fixed set memory locations (20) and marked all their access events as interesting. However, this approach proved insufficient for detecting bugs like KAFKA-17379 discussed in Section 6. Real-world concurrency tests involve numerous memory events—the KAFKA-17379 test contained 91 distinct shared memory locations accessed by multiple threads across 2782 different program locations, with only a few related to the failure. Random sampling likely misses these critical memory locations, causing SURW to miss bugs. When we manually marked the relevant state as interesting, SURW successfully identified the failure. This confirms that identifying interesting events is crucial for SURW's bug-finding effectiveness and highlights the need for automated methods to identify these events as future research directions [76].

8 Conclusion

This paper observes that practical concurrency testing of JVM targets requires a careful evaluation of design choices to maximize both the scope of target applications and the effectiveness of finding bugs, quickly. We presented Fray, a new platform for concurrency testing of data-race-free JVM programs. Fray introduces *shadow locking*, a concurrency control mechanism that orchestrates thread interleavings without replacing existing concurrency primitives, while still encoding their semantics to faithfully express the set of all possible program behaviors. Fray identifies a sweet spot in our design trade-off space. Our empirical evaluation demonstrated that Fray is effective at general-purpose concurrency testing, and can find real concurrency bugs in mature software projects. Fray serves as a bridge between concurrency research and software engineering practice—allowing researchers to evaluate algorithms on industrial codebases while giving developers access to state-of-the-art testing techniques.

9 Acknowledgements

This work was supported in-part by an Amazon Research Award and in-part by the National Science Foundation under grants CCF-2120955, CCF-2429384, and CCF-2453432. We thank Ankush Desai for early discussions that inspired this work, as well as Yuvraj Agarwal and Vyas Sekar for feedback on prior versions of the paper. We use Claude AI to generate the state graphs in Figs. 8b and 9b.

10 Data-Availability Statement

We have made available scripts and data to reproduce our evaluation at https://github.com/cmu-pasta/fray-benchmark [9].

References

[1] AWS Labs. 2024. Shuttle: a library for testing concurrent Rust code. https://github.com/awslabs/shuttle.

[2] Baeldung. 2024. Testing Multi-Threaded Code in Java. https://www.baeldung.com/java-testing-multithreaded. May 2024 version archived at https://web.archive.org/web/20240515050437/https://www.baeldung.com/java-testing-multithreaded.

[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *Proceedings of the 40th international conference on software engineering*. 433–444.

[4] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 167–178.

[5] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*. 2325–2342.

[6] Jong-Deok Choi and Harini Srinivasan. 1998. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*. 48–59.

[7] Lucas Cordeiro and Bernd Fischer. 2011. Verifying multi-threaded software using SMT-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*. 331–340.

[8] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. 2023. Industrial-Strength Controlled Concurrency Testing for Programs with Coyote. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 433–452.

[9] Fray Developers. 2025. *cmu-pasta/fray-benchmark: Fray Artifact Release*. https://doi.org/10.5281/zenodo.15724289

[10] Kafka Developers. 2025. Kafka Jira Issue Tracker with Keyword Concurrency and Flaky. https://issues.apache.org/jira/browse/KAFKA-18845?jql=project%20%3D%20KAFKA%20AND%20text%20~%20%22flaky%22%20and%20text%20~%20%22concurrency%22%20ORDER%20BY%20created%20DESC.

[11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-bounded scheduling. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 411–422.

[12] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[13] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[14] Apache Foundation. 2023. Tomcat: JasperLoader. https://github.com/apache/tomcat/blob/11.0.0/java/org/apache/jasper/servlet/JasperLoader.java#L80.

[15] Apache Foundation. 2024. Apache Lucene. https://lucene.apache.org/.

[16] Apache Foundation. 2024. KAFKA STREAMS. https://kafka.apache.org/documentation/streams/.

[17] GitHub. 2022. The top programming languages. https://octoverse.github.com/2022/top-programming-languages.

[18] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2023. Snowcat: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 35–51. https://doi.org/10.1145/3600006.3613148

[19] Google. 2024. Guava: Google Core Libraries for Java. https://github.com/google/guava.

[20] Google. 2024. This is Weaver, a framework for writing multi-threaded Unit Tests in Java. https://github.com/google/thread-weaver.

[21] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2023. *The Java® Language Specification—Java SE 21 Edition*. Oracle America, Inc., Chapter 17.4 (Memory Model), 761–776.

[22] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2023. *The Java® Language Specification—Java SE 21 Edition*. Oracle America, Inc., Chapter 17 (Threads and Locks), 755–782.

[23] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[24] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 207–216.

[25] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov. 2011. Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 223–233. https://doi.org/10.1145/2025113.2025145

[26] Dae R. Jeong, Yewon Choi, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2024. OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 229–248. https://doi.org/10.1145/3694715.3695944

[27] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[28] Jetbrains. 2025. MonitorTransformers.kt. https://github.com/JetBrains/lincheck/blob/eb9d56e96a732cbe69e7f303924da4849e320efd/jvm-agent/src/main/org/jetbrains/kotlinx/lincheck/transformation/transformers/MonitorTransformers.kt#L36.

[29] Yanyan Jiang, Tianxiao Gu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2014. CARE: Cache guided deterministic replay for concurrent Java programs. In *Proceedings of the 36th International Conference on Software Engineering*. 457–467.

[30] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.

[31] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An extensible active testing framework for concurrent programs. In *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*. Springer, 675–681.

[32] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. *ACM Sigplan Notices* (2009).

[33] Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. 2023. Lincheck: A practical framework for testing concurrent data structures on JVM. In *International Conference on Computer Aided Verification*.

[34] Elastic Search Labs. 2025. Concurrency bugs in Lucene: How to fix optimistic concurrency failures. https://www.elastic.co/search-labs/blog/concurrency-bugs-lucene-debugging.

[35] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 312–322.

[36] Ao Li. 2025. Fray Debugger Plugin for Jetbrains IDEs. https://plugins.jetbrains.com/plugin/26623-fray-debugger.

[37] Ao Li, Byeongjee Kang, Vasudev Vikram, Isabella Laybourn, Samvid Dharanikota, Shrey Tiwari, and Rohan Padhye. 2025. Fray: An Efficient General-Purpose Concurrency Testing Platform for the JVM (Extended Version). arXiv:2501.12618 [cs.PL] https://arxiv.org/abs/2501.12618

[38] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 162–180.

[39] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A benchmark suite of real-world Java concurrency bugs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 178–189.

[40] Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2017. A volatile-by-default JVM for server applications. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 49 (oct 2017), 25 pages. https://doi.org/10.1145/3133873

[41] Tongping Liu, Charlie Curtsinger, and Emery D Berger. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 327–336.

[42] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*.

[43] Shan Lu, Soyeon Park, and Yuanyuan Zhou. 2011. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Transactions on Parallel and Distributed Systems* 23, 6 (2011), 1060–1072.

[44] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices* 42, 6 (2007), 446–455.

[45] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs.. In *OSDI*, Vol. 8.

[46] Robert O'Callahan. 2016. Introducing rr Chaos Mode. https://robert.ocallahan.org/2016/02/introducing-rr-chaos-mode.html.

[47] Robert O'callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 167–178.

[48] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.

[49] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 97–108.

[50] OpenJDK. 2024. Java Concurrency Stress tests. https://openjdk.org/projects/code-tools/jcstress/. Source repository at https://github.com/openjdk/jcstress.

[51] Oracle. 2009. Class Loader API Modifications for Deadlock Fix. https://openjdk.org/groups/core-libs/ClassLoaderProposal.html.

[52] Oracle. 2023. Class java.lang.Thread. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Thread.html#join(long).

[53] Oracle. 2023. LockSupport.parkNanos(). https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/LockSupport.html#parkNanos(long).

[54] Oracle. 2023. Package java.util.concurrent. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/package-summary.html.

[55] Oracle. 2023. Package java.util.concurrent.atomic. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/atomic/package-summary.html.

[56] Chang-Seo Park and Koushik Sen. 2008. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 135–145.

[57] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–74.

[58] David Schwartz, Ankith Kowshik, and Luís Pina. 2024. Jmvx: Fast Multi-threaded Multi-version Execution and Record-Replay for Managed Languages. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 329 (Oct. 2024), 29 pages. https://doi.org/10.1145/3689769

[59] Koushik Sen. 2007. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. 323–332.

[60] Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 11–21.

[61] Koushik Sen. 2015. CalFuzzer deterministic scheduler. https://github.com/ksen007/calfuzzer/blob/4cab1bc162faf17e950027937071a3c19c318a9b/src/javato/activetesting/deterministicscheduler/ApproxDeterministicScheduler.java#L195.

[62] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71.

[63] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake it! detecting flaky tests caused by concurrency with shaker. In *2020 IEEE International Conference on Software Maintenance and Evolution*.

[64] Stack Overflow. 2023. Developer survey—Most popular technologies: Programming, scripting, and markup languages. https://survey.stackoverflow.co/2023/#technology-most-popular-technologies.

[65] Paul Thomson. 2016. *Practical systematic concurrency testing for concurrent and distributed software.* Ph. D. Dissertation. Imperial College London.

[66] Paul Thomson, Alastair F Donaldson, and Adam Betts. 2016. Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing (TOPC)* 2, 4 (2016), 1–37.

[67] TIOBE Software. 2024. TIOBE Index. https://www.tiobe.com/tiobe-index/. July 2024 Index archived at https://web.archive.org/web/20240709010804/https://www.tiobe.com/tiobe-index/.

[68] Tokio. 2024. Loom: Concurrency permutation testing tool for Rust. https://github.com/tokio-rs/loom.

[69] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model checking programs. *Automated software engineering* 10 (2003), 203–232.

[70] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. 13 pages. https://doi.org/10.1145/3510003.3510178

[71] Dylan Wolff, Zheng Shi, Gregory J Duck, Umang Mathur, and Abhik Roychoudhury. 2024. Greybox Fuzzing for Concurrency Testing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 482–498.

[72] Zhemin Yang, Min Yang, Lvcai Xu, Haibo Chen, and Binyu Zang. 2011. ORDER: Object centric deterministic replay for Java. In *2011 USENIX Annual Technical Conference*.

[73] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 485–502.

[74] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing.. In *USENIX Security Symposium*. 2849–2866.

[75] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial order aware concurrency sampling. In *30th International Conference Computer Aided Verification (CAV'18)*.

[76] Huan Zhao, Dylan Wolff, Umang Mathur, and Abhik Roychoudhury. 2025. Selectively Uniform Concurrency Testing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1003–1019. https://doi.org/10.1145/3669940.3707214

Received 2025-07-21; accepted 2025-08-12