Article

Not peer-reviewed version

Black-Box Bug-Amplification for Multithreaded Software

Yeshayahu Weiss [*,†] , Gal Amram [‡] , Achiya Elyasaf [‡] , Oded Margalit [‡] , Eitan Farchi [‡] , Gera Weiss [‡]

Posted Date: 30 July 2025

doi: 10.20944/preprints202507.2304.v1

Keywords: concurrency bugs; bug reproduction; rare event detection; model-based testing; regression modeling; search-based software testing; black-box testing; ensemble methods; noise-tolerant learning; probabilistic bug-amplification

Article

Black-Box Bug-Amplification for Multithreaded Software

Yeshayahu Weiss [1,*,†], **Gal Amram** [2,‡], **Achiya Elyasaf** [3,‡], **Eitan Farchi** [2,‡], **Oded Margalit** [1,‡] **and Gera Weiss** [1,‡]

[1] Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel
[2] IBM Research, Haifa, Israel
[3] Department of Software and Information Systems Engineering, Ben-Gurion University of the Negev, Be'er Sheva, Israel
[*] Correspondence: weissye@post.bgu.ac.il
[†] Current address: Ben-Gurion University of the Negev, Be'er Sheva, Israel.
[‡] These authors contributed equally to this work.

Abstract

Bugs, especially those in concurrent systems, are often hard to reproduce because they manifest only under rare conditions. Testers frequently encounter failures that occur only under specific inputs, even when occurring with low probability. We propose an approach to systematically amplify the occurrence of such elusive bugs. We treat the system under test as a black-box and use repeated trial executions to train a predictive model that estimates the probability of a given input configuration triggering a bug. We evaluate this approach on a dataset of 17 representative concurrency bugs spanning diverse categories. Several model-based search techniques are compared against a brute-force random sampling baseline. Our results show that an ensemble of regression models can significantly increase bug occurrence rates across nearly all scenarios, often achieving an order-of-magnitude improvement over random sampling. The contributions of this work include: (i) a novel formulation of bug-amplification as a rare-event regression problem; (ii) an empirical evaluation of multiple techniques for amplifying bug occurrence, demonstrating the effectiveness of model-guided search; and (iii) a practical, non-invasive testing framework that helps practitioners expose hidden concurrency faults without altering the internal system architecture.

Keywords: concurrency bugs; bug reproduction; rare event detection; model-based testing; regression modeling; search-based software testing; black-box testing; ensemble methods; noise-tolerant learning; probabilistic bug-amplification

1. Introduction

Bugs that manifest nondeterministically, sometimes referred to as *Heisenbugs* [1] or intermittent bugs [2], pose a significant challenge for debugging and validation in complex software systems. This difficulty is particularly pronounced for *concurrency bugs*, which typically arise only under rare thread interleavings or delicate timing conditions. In practice, developers rely on techniques such as manual code inspection and brute-force stress testing to uncover such failures. Although stress testing may occasionally expose these elusive faults, it offers no guarantees of detecting and often fails to detect bugs that appear only under constrained conditions. As a consequence, critical concurrency issues can remain unresolved for extended periods, undermining confidence in the system's reliability.

In this paper, we *maximize the empirical failure probability* observed during testing, under a fixed execution budget. We refer to this goal as *bug amplification*. In contrast to approaches such as rare-event simulation [3] and statistical model checking [4] which typically rely on internal instrumentation, white-box knowledge, or formal specifications; our method operates in a fully black-box manner. We assume no access to source code or internal system behavior. Instead, we systematically vary input parameters, such as workload configurations and timing-related settings, to increase the likelihood that a latent bug will manifest during execution.

Despite progress in the field, reliably exposing concurrency bugs in real-world systems remains an open challenge [5]. Systematic concurrency testing tools attempt to exhaustively explore possible thread schedules and can enable deterministic replay of bugs once discovered. However, their scalability is limited by the combinatorial explosion of scheduling interleavings. Alternatively, randomized scheduling introduces noise to execution timing and has shown improved coverage compared to naive stress tests [6,7], yet remains fundamentally probabilistic and may still miss deeply hidden bugs. Record-and-replay tools log nondeterministic events during execution for later reproduction, but their performance overhead and requirement for tightly controlled environments make them impractical in many settings. Collectively, these approaches fall short of providing a general, scalable solution for reliably triggering elusive concurrency failures.

To address this gap, we introduce a novel approach that frames bug amplification as a black-box optimization problem over the system's input space. Rather than modifying internal code or instrumenting it, we run the system repeatedly under different input configurations and observe whether a failure occurs. These outcomes are used to train a predictive model estimating the probability of failure as a function of the input parameters. The model then guides the generation of future test inputs, focusing resources on regions of the input space more likely to expose the bug.

Casting this task as a regression problem presents unique difficulties. The target function, a binary indicator of failure, is one with an extremely sparse positive signal, often yielding zero in most regions of the input space. Even for failure-prone configurations, the bug may only appear with low probability due to nondeterministic execution. To cope with this challenge, we perform multiple trials for each input and use the average failure rate as a noisy estimate of its true failure probability. This allows us to apply regression algorithms despite the underlying stochasticity and imbalance, though it necessitates robust modeling techniques capable of tolerating label noise and extreme skew.

To evaluate the proposed strategy, we curated a benchmark of 17 concurrency bugs spanning a comprehensive taxonomy of bug symptoms and their underlying causes. These bugs, drawn from real-world and synthetic sources, cover a variety of symptoms (e.g., deadlocks, crashes, data races) and underlying causes (e.g., incorrect synchronization, ordering violations). For each problem, we identified key input parameters that influence bug manifestation and tuned the system so that failures occur with low probability under default settings. This controlled setup enables rigorous assessment of amplification techniques under realistic yet challenging conditions.

We applied several model-based search techniques to the benchmark, including linear regression, decision trees, and nonlinear ensemble methods, and compared them against a baseline of brute-force random sampling. Under identical budget constraints, a stacked ensemble of classifiers consistently achieved the best overall performance, substantially increasing bug manifestation rates across the majority of scenarios.

This work makes the following contributions:

- **Benchmark and Problem Formulation:** We introduce a curated dataset of 17 concurrency bugs and frame the failure-triggering task as a regression problem with sparse positives and stochastic labels—posing distinct challenges for conventional learners.
- **Evaluation of Amplification Techniques:** We systematically compare several model-guided search strategies and show that ensemble-based learning significantly improves bug-triggering probability within practical testing budgets.
- **Practical, Black-box Testing Framework:** Our approach treats the system under test as a black box, requiring no code changes or instrumentation, making it readily applicable in real-world testing workflows.

The remainder of this paper is organized as follows. Section 2 reviews the state of the art in bug reproduction, presenting leading techniques and key challenges in the field. Section 3 provides a detailed classification of concurrency bug types relevant to our study. Section 4 summarizes the benchmark problems used in our evaluation, outlining the criteria for selection. Section 5 describes the core research methods, focusing on the modeling of interleaving in multithreaded code. Section 6 introduces the four bug-amplification methods that we developed and applied, and provides implementation and configuration specifics. Section 7 presents the experimental results and discusses their implications. Finally, the paper concludes with a summary of findings, a detailed list of the limitations of our approach, and directions for future research.

2. State of the Art in Bug Reproduction

Reproducing nondeterministic concurrency failures remains a central challenge in software testing. These bugs typically occur only under rare thread interleavings or specific combinations of environmental and input parameters, making them elusive and difficult to diagnose [8].

Traditional techniques such as stress testing, heuristic scheduling perturbations, and detailed logging have been widely used in practice, but they offer no guarantees and are often insufficient for reliably exposing such rare failures [9]. CARDSHARK [10], for example, demonstrates how even kernel-level bugs may remain unstable without explicit noise control or scheduling alignment.

Industry Practice. When developers encounter rare failures in production, a common response is to attempt reproduction via repeated testing under varied conditions, manipulating input sizes, concurrency levels, or hardware settings [11]. Logging may provide diagnostic clues, but even lightweight instrumentation, such as coverage or profiling hooks, can perturb timing behavior enough to mask or induce concurrency bug manifestation [12]. Kernel-level concurrency testing frameworks such as the eBPF-based technique by Xu et al. [13] offer promising lightweight instrumentation for observing concurrency bugs in real-world deployments.

Systematic Exploration. Research tools such as CHESS [14], Nekara [15], and Fray [5] aim to improve bug reproducibility by exhaustively exploring thread schedules in bounded spaces. CHESS is a pioneering systematic testing tool for multithreaded Windows applications that explores all interleavings under a given bound. Nekara is an open-source, cross-platform library (2021) that enables developers to define semantics for concurrency primitives and systematically explore schedules in a controlled, repeatable manner. Fray, introduced in 2025, offers efficient black-box schedule control and instrumentation for JVM-based systems. These tools can replay discovered interleavings deterministically, a key advantage for debugging, but they require either source or binary instrumentation and do not scale well with large programs or vast input spaces.

Probabilistic Scheduling and Sampling. Techniques like Probabilistic Concurrency Testing (PCT) [16], iterative schedule fuzzing [17], and directional scheduling of synchronization primitives in Go programs [18] attempt to bias execution toward schedules more likely to reveal bugs. While these methods can improve exposure rates, they remain largely unguided by feedback from prior executions.

Learning-Based Approaches. Recent advances have begun exploring machine learning for bug localization and input generation [19,20], but most treat the system as a white box or focus on symbolic execution, mutation, or coverage estimation [21]. By contrast, our method treats the system as a black-box and explicitly aims to maximize the empirical failure probability via predictive models over the input domain.

Building on these advances, our work treats bug reproduction as a noisy optimization problem over inputs, training predictive classifiers to guide search. Instead of exploring schedules, we vary inputs and use learned models to amplify bug occurrence rates within constrained testing budgets, improving reproducibility and efficiency.

3. Types of Concurrency Bugs

Following prior work such as [22], we introduce a taxonomy to support the evaluation of our bug-amplification techniques. This taxonomy classifies concurrency bugs along two orthogonal dimensions: *observable effect* and *root cause*.

In detail, the observable effect axis captures how a concurrency bug manifests at runtime, i.e., the observable effect or symptom from the system's perspective. The second or root cause axis reflects the underlying cause of the failure, identifying the specific logic error or design flaw in the program's synchronization or concurrency control.

Classification of the observable effects of the concurrent bug is done using the following categories.

- **Deadlock:** A system state in which two or more threads are indefinitely blocked, each waiting for a resource that will never become available, e.g., because it is held by another. The system halts and cannot make further progress.
- **Unexpected Data:** Shared variables take on incorrect or inconsistent values due to unsynchronized access, race conditions, or improper interleaving of reads and writes.
- **Concurrent Access:** Multiple threads enter a critical section simultaneously, violating mutual exclusion and potentially corrupting shared state or breaking invariants.

While the classification of the root cause of the concurrent bug is done using the following categories.

- **Missing or Weak Guarding:** Inadequate protection of critical sections, often due to absent atomicity checks, incorrect condition synchronization, or overreliance on scheduling assumptions.
- **Non-Atomic Operations on Shared State:** Access to shared data is implemented via sequences of non-atomic operations, allowing interleaving by other threads to interfere with correctness.
- **Incorrect Command Ordering:** Synchronization operations are issued in the wrong order, violating required temporal constraints. For example, a thread signals a condition before another begins waiting for it.
- **Misuse of Concurrency Primitives:** synchronization constructs such as locks, semaphores, and condition variables are used incorrectly, e.g., in unintended contexts, or in ways that violate their semantics.

The cross-product of these two axes yields twelve distinct categories of concurrency bugs, each representing a unique pairing of effect and cause. Table 1 summarizes the distribution of our benchmark problems across this taxonomy, with each problem assigned to the cell corresponding to its observed effect and inferred root cause. As a root cause may have more than a single effect, a problem index may appear twice in the same column, but not in the same row.

Table 1. Classification of concurrency problems by *Effect* (rows) and *Root Cause* (columns), showing the problem number and name. Note that some problems may produce multiple effects (e.g., Problems 12 and 4).

Effect \ Root Cause	Missing/Weak Guard	Non-Atomic Op.	Incorrect Ordering	Misuse of Primitives
Deadlock	6 (If-Not-While) 8 (Lost Signal) 17 (Sleeping Guard)	11 (Race-To-Wait)	7 (Lock Order Inversion) 16 (Signal-Then-Wait)	2 (Broken Barrier) 5 (Flagged Deadlock)
Unexpected Data	6 (If-Not-While) 9 (Partial Lock)	12 (Racy Increment) 14 (Shared Counter)	4 (Delayed Write)	1 (Atomicity Bypass)
Concurrent Access	3 (Broken Peterson) 15 (Shared Flag)	12 (Racy Increment) 14 (Shared Counter)	4 (Delayed Write)	10 (Phantom Permit) 13 (Semaphore Leak)

The inclusion of at least one benchmark problem in each of the twelve cells of the classification matrix ensures that our taxonomy is comprehensively represented. This guarantees that the analysis spans all combinations of observable effects and root causes, ensuring broad and representative coverage of concurrency failure modes.

4. Summary of the Benchmark Problems

To evaluate our ability to amplify and detect failure cases in multithreaded systems, we assembled a benchmark that spans the primary classes of concurrency faults. Each problem instance illustrates a distinct error pattern, and the accompanying description clarifies the type of defect it represents. The benchmark is available in a *GitHub repository*[1]

The benchmark is based on the canonical puzzles from *The Deadlock Empire*[2], an interactive collection of multithreading challenges that can be executed step-by-step. To achieve the broader coverage outlined in the previous section, we extended this initial set with additional cases gathered from the literature and custom-crafted variants, until all combinations of Effect (Deadlock, Unexpected Data, Concurrent Access) and Root Cause (Missing or Weak Guarding, Non-Atomic Operations, Incorrect Command Ordering, Misuse of Concurrency Primitives) were represented.

Section 9 provides a full description of each of the 17 concurrency problems enumerated below. For every problem, we explicitly document (i) the scenario, (ii) its observable effect, (iii) the underlying root cause according to our taxonomy, and (iv) a concise insight that summarizes the key lesson. This curated collection provides a balanced testbed for assessing failure-amplification techniques across the full spectrum of concurrency bugs.

Atomicity Bypass: A thread releases a lock before completing a read-modify-write, leading to data corruption despite apparent locking. See Section 9.1.

Broken Barrier: Improper barrier reuse or reset causes some threads to wait forever, expecting others to arrive. See Section 9.2.

Broken Peterson: Incorrect implementation of Peterson's algorithm allows both threads to enter the critical section. See Section 9.3.

Delayed Write: Operations are reordered due to compiler or logic flaws, leading to stale reads or broken invariants. See Section 9.4.

Flagged Deadlock: Threads use flags and spin loops incorrectly, creating interleaving paths that deadlock. See Section 9.5.

If-Not-While: A thread waits using an `if` condition instead of a `while` loop, leading to missed signals and unsafe access. See Section 9.6.

Lock Order Inversion: Classic deadlock: threads acquire two locks in opposite order, causing circular wait. See Section 9.7.

Lost Signal: A thread sends a signal before another begins waiting on a condition variable; the signal is lost, causing a deadlock. See Section 9.8.

Partial Lock: Only part of the critical section is protected by a lock; race conditions still occur. See Section 9.9.

Phantom Permit: A semaphore is released without a corresponding `Wait`, allowing more threads than expected to enter the critical section. See Section 9.10.

Race-To-Wait: Threads race to increment a shared counter and both wait on a condition that never becomes true due to non-atomic updates. See Section 9.11.

Shared Flag: A single boolean flag is used for synchronization without proper mutual exclusion, allowing concurrent access. See Section 9.15

Signal-Then-Wait: A thread signals with `notify_all()` before the other enters the wait; the notification is missed despite a guarded `while` loop. See Section 9.16

Sleeping Guard: A thread goes to sleep on a condition variable without checking the actual shared state, causing missed wakeups and deadlock. See Section 9.17

[1] https://github.com/geraw/bug_amp

[2] https://deadlockempire.github.io/#menu

doi:10.20944/preprints202507.2304.v1

Listing 1: Core simulation loop controlling the execution of multiple threads. Each thread yields a delay, and the scheduler selects the next thread to execute based on wake-up times.

```
def simulate(_threads, init=lambda:None, init_arg=None, expected_invariant=None):
    init(init_arg)                          # Initialize global variables
    gen = [t() for t in _threads]           # Create generators (threads)
    wake_times = [0] * len(_threads)        # Initial wake times
    while any(t < END for t in wake_times):
        nxt = np.argmin(wake_times)         # Select next thread to wake
        wake_times[nxt] += next(gen[nxt])   # Advance its wake time
        if expected_invariant is not None:
            assert expected_invariant()     # Check system invariant
```

5. Interleaving Multithreaded Code

In this section, we describe our method for simulating multithreaded programs in a controlled and repeatable manner using Python generators. To enable systematic exploration and direct comparison across a variety of concurrency scenarios, we adopt a uniform representation strategy that brings clarity, modularity, and flexibility to our simulation framework.

Each problem is encoded as a collection of Python generator functions. Each generator models a single thread that operates on the System Under Test (SUT) and uses `yield` statements to explicitly mark points where execution may pause and control may be transferred to another thread. Modeling representation allows us to canonize a wide range of concurrency scenarios into a common format, facilitating repeatable experiments and meaningful comparisons under different timing conditions. Our framework further incorporates parameter-dependent delays, which can include both structured variation (e.g., based on thread-specific parameters or environment emulation) and random noise. This enables modeling of both deterministic scheduling and nondeterministic, real-world variability.

Together, these design choices provide a robust and extensible foundation for simulating complex concurrency behaviors and analyzing how timing-related parameters influence system correctness. The types of problems we address typically involve multiple threads, shared variables, and bugs that are triggered only under specific interleavings, often governed by subtle timing conditions. To simulate such behavior, we employ the `simulate()` function shown in Listing 1, which orchestrates the execution of multiple threads according to a parameter-driven timing model.

The `simulate()` function manages a set of thread generators. Each thread yields a value representing how long it wishes to "sleep" (i.e., delay its next execution step), and the simulation engine schedules the threads based on their wake-up times. The thread with the shortest delay is resumed first, simulating a time-based interleaving of execution steps. Importantly, the simulation does not involve real-time waiting or system-level delays. Instead, it operates in virtual time, advancing the logical clock and reordering thread execution based on the declared delays, thereby allowing efficient exploration of possible interleavings without wasting actual runtime.

Each thread is implemented as a generator function that performs a sequence of atomic operations, with `yield` statements marking the boundaries between them. These yield points indicate simulated delays during which other threads may execute. An illustrative example is provided in Listing 2.

This example models a typical concurrency issue: the thread sets a shared variable `x` to a fixed value, but due to interleaved execution, another thread might overwrite it before the current one verifies its value. The timing between steps is simulated by yielding expressions that define how long each thread "sleeps" before proceeding. Each delay expression consists of three components. The first is a global coefficient C, which reflects the overall processing speed or workload of the simulated system. The second is a parameter D_i, representing the nominal delay associated with a specific operation. The third component is a call to `distortion()`, which introduces random variation to simulate environmental unpredictability such as jitter or fluctuating system load.

Listing 2: A thread modeled as a generator. Yields represent delays between atomic steps. The delays depend on a system-wide coefficient C and problem-specific parameters D_i, with optional noise added.

```
def simulated_thread():
    global x                            # Shared variable
    for i in range(10):
        yield C * D1 + distortion()     # Simulated delay
        x = 3                           # Atomic operation
    yield C * D2 + distortion()         # Simulated delay
    if x != 3:
        yield C * D3 + distortion()     # Additional delay before assert
        assert x != 3                   # Bug condition
    yield END
```

This parametrization enables the simulation to model a wide range of execution environments and conditions. By adjusting the coefficient C, we can emulate machines with varying processing speeds or scheduling overhead. Changing the D_i values allows us to control the logical duration of specific computation segments. The addition of noise via `distortion()` allows us to explore nondeterministic interleavings, helping to uncover rare or timing-sensitive bugs that would otherwise be difficult to reproduce.

Simulating Rare Failures: Many concurrency bugs, especially those related to race conditions and ordering violations, are notoriously difficult to reproduce in real systems because they manifest only under rare timing conditions. Our simulation framework addresses this challenge by treating delay parameters as inputs. Specifically, each test-case accepts a tuple of values representing delays (e.g., D_1, D_2, D_3), and runs the simulation multiple times using different random seeds for `distortion()`. Each simulation run returns a result indicating whether a failure (e.g., assertion violation) occurred. By aggregating the outcomes across many runs, we can estimate the probability that a particular configuration of delays leads to a bug. This approach is especially useful for identifying critical thresholds or delay combinations that increase the failure likelihood.

Invariant Checking: Optionally, a predicate `expected_invariant` can be passed to the `simulate()` function. This predicate is evaluated after each execution step to ensure that the system remains in a valid state. Violations of this invariant are treated as test failures and help pinpoint scenarios of the manifestation of concurrency bugs.

5.1. Evaluation Protocol

To enable a fair, consistent, and statistically robust evaluation of the bug-amplification methods under study, we define a controlled experimental framework that governs how test-cases are generated, evaluated, and compared. This framework incorporates a fixed execution budget, multiple randomized trials, and an analysis of the top-performing test-cases across different metrics. Together, these components ensure that our assessment is not only reproducible but also reflective of real-world usage scenarios such as iterative debugging and automated fault localization pipelines.

Budget Consumption. Each bug-amplification method is constrained to a fixed execution budget of B runs of the SUT. The budget is progressively consumed in increasing blocks of test-case numbers, $n \in \{100, 300, \ldots, 3900\}$, allowing each method to iteratively improve its selection while avoiding early resource exhaustion. At each checkpoint, the accumulated executions are analyzed to update the observed probability of bug exposure. This staged consumption strategy supports convergence analysis and ensures that all methods operate under identical cost constraints while striving to maximize effectiveness. The specific mechanisms by which each method adheres to this budget constraint are detailed in their respective descriptions, and an overview of the budget split across iterations is provided in Table 2.

Table 2. Budget allocation per methods. **brute-force (*BF*)** spends the first B/k runs, where k is minimum repeat size, on *estimation* of a random candidate and immediately reports that score; there is no exploitation phase. **Simulated Annealing (*SA*)** divides the budget into steps s and neighborhood size k , enabling explicit control of SUT invocations. The **Genetic Algorithm (*GA*)** uses a population of $k = 50$ and evolves for B/k generations. The **Ensemble classifier (*Ens*)** devotes the entire budget to model-guided search: at every step, it samples 100 random inputs (exploration) and 100 model-ranked inputs (exploitation), retrains, and repeats.

Method	Exploration	Exploitation	Notes
BF	B/k random	-	k is the minimum repetition required
SA	k per step	B/k steps	
GA	pop. k per generation	B/k generations	
Ens	add 100 random/iter	add 100 ranked/iter	Full budget trains model each iter

Nonetheless, while strict budget adherence is maintained throughout each method's execution, the final evaluation phase in this study deliberately exceeds these constraints. This extended phase is not part of the methods themselves, but is introduced solely for the purpose of research evaluation. Specifically, to rigorously assess the quality of the selected test-cases, typically those with the highest observed failure likelihood, we subject each case to massive re-execution with the SUT, far beyond the original budget. This allows us to derive a precise and statistically robust estimate of its true failure-inducing potential.

Repeated trials. To obtain statistically meaningful estimates, each $\langle method, problem \rangle$ pair was executed 50 independent times. Each run exploited the full budget schedule above, producing a *single* best-scoring test-case, i.e., the input with the highest observed failure probability. Aggregating the best scores over 50 runs yields the sample mean and standard deviation that appear in all result plots.

Top-*k* Analysis. In practice, automated debugging pipelines require three disjoint pools: development (*debugging*), model training (*testing*), and final assessment (*validation*). Reporting only the single best-case risks overfits, whereas presenting the entire budget is often impractical. Hence, we also study the 5th and 10th best inputs, providing a small yet diverse set that effectively supports such a pipeline.

6. Bug-Amplification Methods

This section presents the various search techniques explored in our study to amplify the probability of detecting concurrency bugs. The generation of effective test-cases presents both a statistical and an algorithmic challenge. Our goal is to investigate whether advanced heuristics can outperform naive or exhaustive methods in this context. Each subsection below introduces a distinct test generation paradigm, ranging from brute-force enumeration to learning-based classification, and describes its design, rationale, and implementation as applied to our concurrency benchmark.

6.1. Baseline: Random Search

Random search serves as the baseline method in this study, providing a critical comparison point for evaluating the effectiveness of more sophisticated search techniques. This method operates without incorporating any domain-specific heuristics or optimization strategies, offering conceptual simplicity and ease of implementation. Its role is to help determine whether complex methods are truly necessary, or if random exploration is sufficient for discovering high-probability failure-inducing test-cases.

The process begins by randomly generating (B/k) candidate test-cases. Each candidate is evaluated by executing it multiple (k) times against the system under test, in order to estimate its likelihood of triggering a bug. Every execution yields a binary outcome – failure (bug-triggered) or success. A scenario's estimated score is calculated as the frequency of failures across its executions, with the number of repetitions serving as a configurable parameter that trades evaluation accuracy for computational cost.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 30 July 2025 doi:10.20944/preprints202507.2304.v1

We used a fixed sampling parameter $k = 30$ for each test-case. This value was chosen based on the statistical justification provided by the Law of Large Numbers (LLN) and the Central Limit Theorem (CLT), which suggest that 30 independent samples are generally sufficient to obtain a stable estimate of the mean and variance. This ensures that the bug exposure probability computed from the k test-cases is both statistically meaningful and computationally efficient.

Like all tested methods, random search operates within a fixed execution budget B as described earlier. Once the budget is consumed, candidates are ranked by their estimated failure probability, and the top-ranked scenarios are selected as the method's output. Throughout the remainder of the paper, we refer to this approach as the Brute-Force (BF) method.

In the Results section, we examine scenarios where advanced search methods offer clear benefits and compare them with cases where the BF method performs adequately.

6.2. Simulated Annealing

Finding a concurrency bug in a continuous search space can be viewed as climbing an unknown, locally smooth *probability landscape* $p(x)$ whose height at a point $x \in \mathbb{R}^n$ represents the likelihood that the corresponding test input triggers the fault. Our Simulated-Annealing (SA) variant explores this landscape by iteratively sampling a small neighborhood of the current point and then moving in the direction where failures are more concentrated.

Why this variant? We developed this SA variant for three reasons. (i) **Budget control:** By fixing k candidates per step ($k = 30$ as described in 6.1) and s ($s = B/k$) optimization steps, we guarantee an exact run budget B. Most generic SA frameworks expose only the iteration count and can silently overshoot the allowed SUT executions. (ii) **Noise awareness.** Each fitness evaluation is stochastic, so the algorithm must cope with noisy measurements, a feature rarely found in off-the-shelf SA libraries. (iii) **Geometric clarity.** The center-of-mass update rule (see Figure 1) offers an intuitive, easily inspectable implementation that has proven effective in practice.

Neighbourhood sampling. Let $u \in \mathbb{R}^n$ be the current input vector. We draw k random candidates $\{x_1, \ldots, x_k\}$ from the ball $B(u, \varepsilon) = \{x \mid \|x - u\| \leq \varepsilon\}$. After executing each candidate, we label it *positive* (P) if it triggers the bug $(p(x_i) = 1)$ and *negative* (N) otherwise $(p(x_i) = 0)$.

Center-of-mass estimate. We summarize the neighborhood by the averages

$$\mathbf{N} = \frac{1}{|\mathrm{N}|} \sum_{x_i \in \mathrm{N}} x_i, \qquad \mathbf{P} = \frac{1}{|\mathrm{P}|} \sum_{x_i \in \mathrm{P}} x_i,$$

which act as coarse estimates of where the bug is less (**N**) or more (**P**) likely to occur.

Update rule. Intuitively, we wish to move away from the negatives and, if positives exist, steer toward their center of mass. We therefore (i) take a step from u opposite to **N** and (ii) if positives are present, average that tentative step with **P**'s position. The construction is illustrated in Figure 1, and a Python sketch appears in Listing 3.

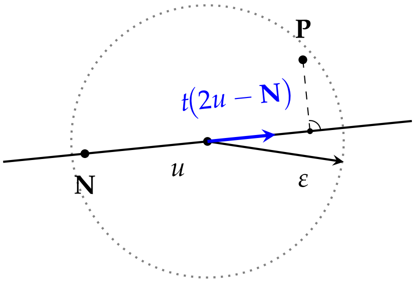

Figure 1. Geometric intuition in 2D of the update step. The k candidates are sampled inside the dotted ball $B(u, \varepsilon)$ centered at u. We move from u to $u_{\text{next}} = t(2u - \mathbf{N})$ (blue arrow) and, if positives exist, bias the step toward the positive center **P**. The dashed segment from **P** is perpendicular to the search line. The radius of the sampling ball is marked by ε.

Listing 3: An implementation of Neighbourhoods Sampling.

```python
def next_point(u, epsilon=0.1, k=30, bounds=[(0,1)]*20):
    # Step 1: Randomly choose k points in the ball B(u, epsilon)
    S = [generate_within_bounds(u, epsilon, bounds) for _ in range(k)]

    # Step 2: Execute each x_i and determine whether the bug was found
    id_x = [run_test(np.array(x_i)) for x_i in S]

    # Step 3: Create two averages N and P
    N = np.mean([x_i for x_i, id in zip(S, id_x) if id == 0], axis=0)
    P = np.mean([x_i for x_i, id in zip(S, id_x) if id == 1], axis=0)

    if P.empty() or N.empty():
        u_next =  S[0]      # arbitrary point
    else:
        # Step 4: Obtain a new point w' and take the average of P and w' as the next point in the search
        w_prime = 2*u - N
        u_next = (P + w_prime) / 2

    return u_next
```

Edge cases: If no positive points or no negative points are found, we choose u_{next} as a random point within $B(u, \varepsilon)$. As ε gradually decreases (the usual annealing schedule), the search converges on increasingly precise regions of high failure probability while still escaping unpromising basins.

6.3. Genetic Algorithm-Based Search

To explore failure-inducing test-cases, we employed a Genetic Algorithm (*GA*) using the EC-KitY evolutionary computation framework [23]. The goal of the algorithm is to evolve test inputs that are likely to trigger failures in a concurrent system, guided by a fitness function that reflects the probability of failure.

We configured the *GA* with a population size of $k = 50$ individuals per generation. The total number of generations is determined by dividing the available test-cases budget by the population size, ensuring that each individual is evaluated once per generation. Fitness is computed using a user-defined `BugHuntingEvaluator`, which estimates the likelihood of a bug manifesting during execution. Since this is a maximization task, higher fitness indicates more failure-prone test-cases. Each individual is represented as a real-valued vector constrained within predefined bounds (depending on each problem).

We tuned the genetic algorithm's hyperparameters to balance convergence speed and search diversity while staying within our evaluation budget. To that end, we selected a population size of $k = 50$, following classical guidelines by Goldberg [24] and more recent studies [25] that recommend sizes in the range of 30–100 to ensure sufficient diversity without excessive cost. We used a two-point crossover with a 0.5 probability to promote recombination of substructures, and uniform mutation applied to 10 randomly selected components with a 0.15 probability to inject controlled variation. Tournament selection with a size of four was chosen to provide moderate selective pressure while preserving population diversity. These values were selected based on standard practice and empirical effectiveness in evolutionary search.

The *GA* employs the following operators:

- **Crossover:** A two-point crossover (`VectorKPointsCrossover`) with a probability of 0.5 exchanges two genome segments between parent individuals. This promotes the recombination of useful substructures and accelerates convergence.

- **Mutation:** Uniform N-point mutation (`FloatVectorUniformNPointMutation`) is applied to 10 randomly selected vector components with a probability of 0.15. This introduces variation and helps the population explore new regions in the search space.
- **Selection:** We use tournament selection with a size of four, where the fittest individual among four randomly sampled candidates is chosen as a parent. This balances selective pressure and population diversity.

We applied elitism by retaining the single best individual in each generation, and terminated the run if no improvement was observed in the best fitness over 100 consecutive generations.

The EC-KitY is a modular and extensible evolutionary computation toolkit for Python, designed to support a wide range of evolutionary techniques including genetic algorithms, genetic programming, coevolution, and multi-objective optimization. It also provides seamless integration with machine learning pipelines, particularly via `scikit-learn`.

As part of our investigation into effective methods for test-case generation, we explored using Genetic Programming, based on the hypothesis that dependencies exist among the input parameters of failure-inducing scenarios. Specifically, we considered the possibility of defining a domain-specific language capable of capturing structural patterns and relations between parameters that frequently lead to failures. This hypothesis was inspired by prior work suggesting that, in most cases, only a small subset of input parameters is responsible for triggering bugs [26].

However, despite initial efforts, the genetic programming process failed to converge toward meaningful patterns, and the approach was ultimately abandoned. As a result, we redirected our efforts toward a more conventional search method, utilizing a generic Genetic Algorithm (*GA*) instead.

6.4. Classification-Based Method: Ensemble Stacking Classifier

In this study, we investigated several supervised learning techniques to enhance the identification of failure, inducing test-cases. After evaluating multiple classifiers, including Random Forests and Multilayer Perceptrons, we observed no significant differences in performance between them. In addition, we experimented with several regression-based models, but they failed to provide reliable prioritization of failure-prone test-cases. As a result, we adopted the Ensemble Stacking Classifier as our primary model, leveraging its ability to combine the strengths of various base learners into a unified predictive framework.

Stacking Architecture

Layer 1 comprises four diverse classifiers-Logistic Regression, Decision Tree, Random Forest, and an MLP, each trained independently to return the probability that a test-case triggers a failure. *Layer 2* is a Logistic Regression meta-learner that ingests both the base-model probabilities and the raw input features (via `passthrough = True`). To curb overfitting, the meta-learner is trained with 5-fold cross-validation, using out-of-fold predictions from the base models. Listing 4 presents the Ensemble Stacking Classification implementation, using the most common classifiers together.

Pre-processing: To address the inherent class imbalance in our data, we apply the *Synthetic Minority Over-sampling Technique* (SMOTE) before training. This ensures a balanced representation of failure and non-failure cases, which improves generalization and stabilizes model training. After training, the ensemble classifier assigns a failure probability to each unseen test-case. These predictions are used to rank test-cases, enabling prioritized execution under a limited testing budget, with failure-prone inputs examined first.

Data Preparation Pipeline: To ensure effective training of the ensemble model, we employed a structured data preparation pipeline comprising three integrated phases. The process began with an initialization step, where we seeded the training dataset using test-cases that had previously triggered system failures during the early stages of bug discovery. This provided a foundational set of informative examples for the model to learn from.

Listing 4: The Ensemble Stacking Classifier. After experimentation, we found that stacking the four most common classifiers and combining their predictions using logistic regression gives the best results. We configured `passthrough=True` to allow raw features to reach the meta-learner and `cv=5` for robust out-of-fold training. We also adjusted the number of iterations to cope with model complexity and used `class_weight=’balanced’` due to skewed data, as bugs are rarely triggered. A two-layer neural network with adaptive learning rate further enhances abstraction and generalization.

```
base_learners = [
    (’lr’, LogisticRegression(max_iter=1000, class_weight=’balanced’)),
    (’dt’, DecisionTreeClassifier(class_weight=’balanced’)),
    (’rf’, RandomForestClassifier(n_estimators=100, class_weight=’balanced’)),
    (’mlp’, MLPClassifier(hidden_layer_sizes=(50, 20),
            learning_rate=’adaptive’, max_iter=500, early_stopping=True))
]

meta_learner = LogisticRegression(class_weight=’balanced’,max_iter=1000)

stacked_model = StackingClassifier(
    estimators=base_learners,
    final_estimator=meta_learner,
    cv=5,
    passthrough=True)
```

In the next phase, we extended the dataset through a budget-guided expansion strategy. This included both exploitation and exploration mechanisms: the model was used to identify new test-cases with high predicted failure probabilities (exploitation), while additional test-cases were also sampled randomly (exploration) to ensure input diversity and guard against model bias.

Finally, in the evaluation phase, the trained model was applied to a large pool of previously unseen test inputs. Based on the predicted failure likelihood, we selected the top-ranked cases for exhaustive system execution. This allowed us to assess the actual failure rates of prioritized inputs, independently of the training budget, thereby providing a robust estimate of the model’s predictive utility.

Observations and Rationale: The Ensemble Stacking Classifier consistently demonstrated reliable and accurate predictions across experiments, providing a balanced trade-off between generalization, robustness, and computational feasibility. Its ability to incorporate multiple perspectives from heterogeneous learners contributed to a more stable and accurate prioritization of test-cases. This made it a natural choice as the core classification model in our failure detection framework.

7. Results

This section presents the empirical results obtained from evaluating our test generation framework on a suite of 17 benchmark concurrency problems, each containing a known, seeded bug. To assess effectiveness, we applied four black-box test generation methods: *Brute-Force (BF)*, *Ensemble Classifier (Ens)*, *Genetic Algorithm (GA)*, and *Simulated Annealing (SA)*. Each method was executed in 50 independent trials per problem to mitigate the influence of stochastic variability and enable robust statistical analysis. During each execution, the method generated a unique test suite, and we recorded whether any test-case within it successfully triggered the target bug. For every method–problem pair, we computed the empirical probability of failure discovery, alongside standard deviation and 95% confidence intervals.

The structure of this section follows the types of visualizations used to interpret the results: overall bug-detection rates per problem, convergence behavior as a function of test budget, top-ranked test-case performance comparisons, and statistical significance analyses between methods. Each type of graph is introduced with an explanation of what it conveys, how the data is structured, and what insights emerge from the results.

doi:10.20944/preprints202507.2304.v1

7.1. Overall Success Rates per Problem

This subsection presents the average probability of successfully triggering each bug using the four tested methods. Each bar represents the mean probability computed across 50 independent runs for a fixed number of test-cases.

The graph in Figure 2 allows direct comparison of method effectiveness across the 17 problems. As observed for 500, 1100, 2100, and 3900 test-cases, the *Ens* method consistently outperforms the other methods in most cases, often achieving significantly higher success rates with lower variance. The *BF* method generally lags behind, especially on more complex bugs.

Across all 17 benchmarks, the ensemble (*Ens*) already achieves a mean success probability of 0.68 ± 0.06 after the first 500 tests, compare to 0.24 ± 0.05 for *GA*, 0.17 ± 0.04 for *BF*, and only 0.04 ± 0.02 for *SA*; by the full 3,900 test budget these averages rise to 0.87, 0.46, 0.39, and 0.11 respectively. While the ensemble-based method consistently outperforms the other approaches in most configurations, its advantage over *GA* in this instance is less pronounced. Specifically, the comparison yields a one-sided Wilcoxon p-value of 0.048, with a 95% confidence interval of [0.03, 0.41]. These results indicate only marginal evidence of superiority, rather than a substantial widening of performance [27].

This type of visualization provides a macroscopic view of method performance per problem and confirms the robustness of the classifier-based approach.

Figure 3 provides a crucial “bird’s-eye view” of the comparative performance of selected optimization and four searching methods. This high-level summary allows one to quickly grasp the overall landscape of method effectiveness without delving into the intricacies of individual experimental variations. This visualization is generated by processing and aggregating data from all problems’ results. The x-axis represents the number of test-cases, while the y-axis indicates the average probability. From these aggregated curves, key insights can be gleaned, such as the convergence behavior of each method as the number of test-cases increases, their relative performance ceilings, and the efficiency with which they approach optimal solutions.

doi:10.20944/preprints202507.2304.v1

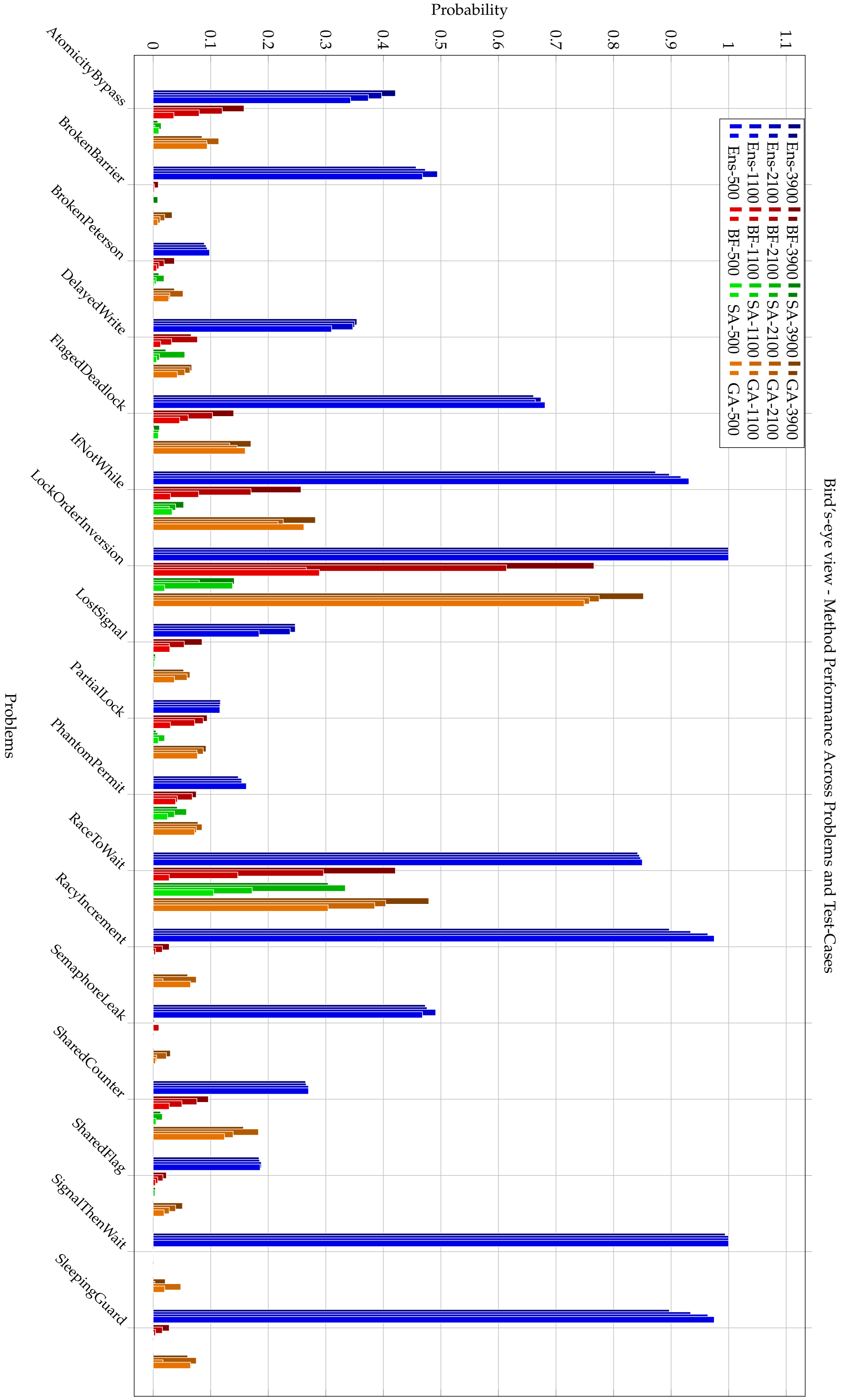

Figure 2. Bird's-eye view for all problems, probability of triggering bug after 500, 1100, 2100, and 3900 test-cases. Each bar is one experiment and based on 50 independent runs. The X axis is all 17 problems, and for each problem, 4 methods and 4 (out of 20) test-cases are shown. The y-axis is the maximum probability for the best test-case.

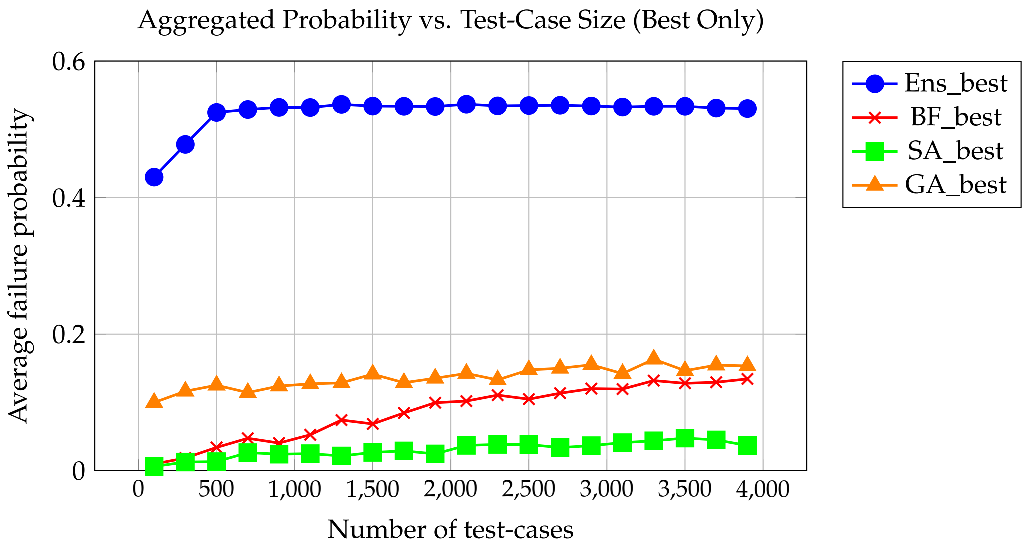

Figure 3. Aggregated performance comparison of four methods across all 17 benchmark concurrency problems. The x-axis shows the number of test-cases used in each evaluation, and the y-axis shows the average fault-triggering probability. For each method, the curve represents the mean of the mean of the best test-case's fault-triggering probability across all problems.

7.2. Per-Problem Bug-Detection Rates

To gain a deeper understanding of method behavior across varying bug difficulties, we divided the 17 benchmark problems into three groups based on their maximum observed bug-detection probabilities: problems with low detectability (maximum probability below 0.2), medium detectability (between 0.2 and 0.6), and high detectability (above 0.6). This classification reflects the intrinsic challenge of each problem and enables structured comparison across problem types.

For illustrative purposes, we present in this section one representative problem from each group. These examples serve to demonstrate trends that consistently appear across the full suite of benchmarks. In all selected cases, the ensemble method (*Ens*) clearly outperforms the alternatives, both in terms of detection probability and convergence rate. The full set of graphs for all 17 problems is included in the supplementary material.

Figure 4 shows representative cases from a low group (Shared Flag), a moderate group (Atomicity Bypass), and a high group (Race-To-Wait). Here, *Ens* rapidly increases its bug-detection success rate, reaching a median of 50% after only 1,000 tests. In contrast, the baseline method (*BF*) lags behind at approximately 15%, while *GA* reaches around 20%. The *SA* method is the least effective, remaining near zero throughout. This pattern, where *Ens* dominates, *GA* and *BF* perform moderately, and *SA* struggles, recurs in nearly all problems, regardless of their detectability group.

Quantitatively, *Ens* exceeds the 0.20 success threshold on 9/9 "low-detectability" problems, while the next best method (*GA*) manages it on only 3; in the medium tier (max $\in [0.2, 0.6]$) *Ens* surpasses 0.60 on 5/6 problems vs. 0 for *BF* and 1 for *GA*; and for high-detectability bugs *Ens* reaches ≥ 0.90 on 4/5 problems within 1100 tests, a level *BF* attains on just one problem.

These results reinforce findings from prior work [28,29], showing that learning-guided techniques not only improve final detection rates, but also significantly reduce the number of test-cases required to reveal faults, particularly in scenarios where failures are elusive or require precise triggering conditions.

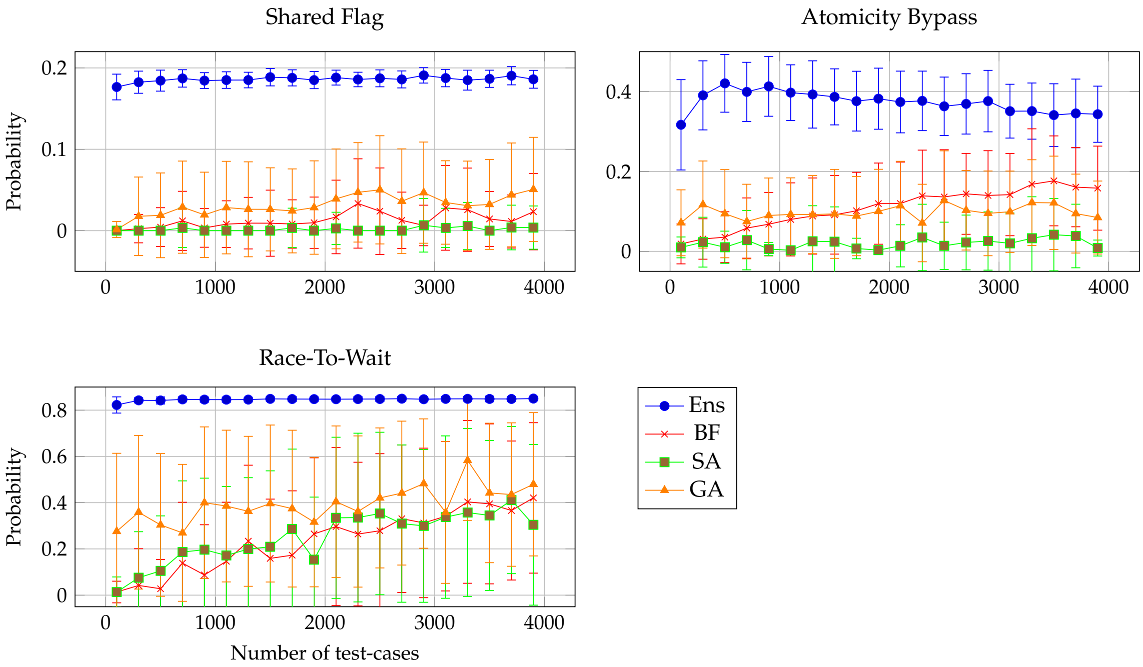

Figure 4. Bug-detection rates across three benchmark problems with different detectability levels. Based on 50 runs; error bars = SD.

7.3. Top-k Case Effectiveness

This section compares the performance of *Ens* compare to the *BF* method when selecting the top-5 and top-10 best test-cases from a larger candidate set. As previously explained, it is often necessary to generate multiple test scenarios for different phases of the process (development, debugging, testing, and validation). Therefore, we demonstrate the ability to generate either the 5th or the 10th test-case according to the two main methods: *Ens* and *BF*. These results, shown in Figure 5 for 5th-best and 10th-best for low detectability (Shared Flag), for medium detectability (Atomicity Bypass); and for high detectability (Race-To-Wait), demonstrate how prioritizing test-cases by a classifier model yields a higher likelihood of bug exposure.

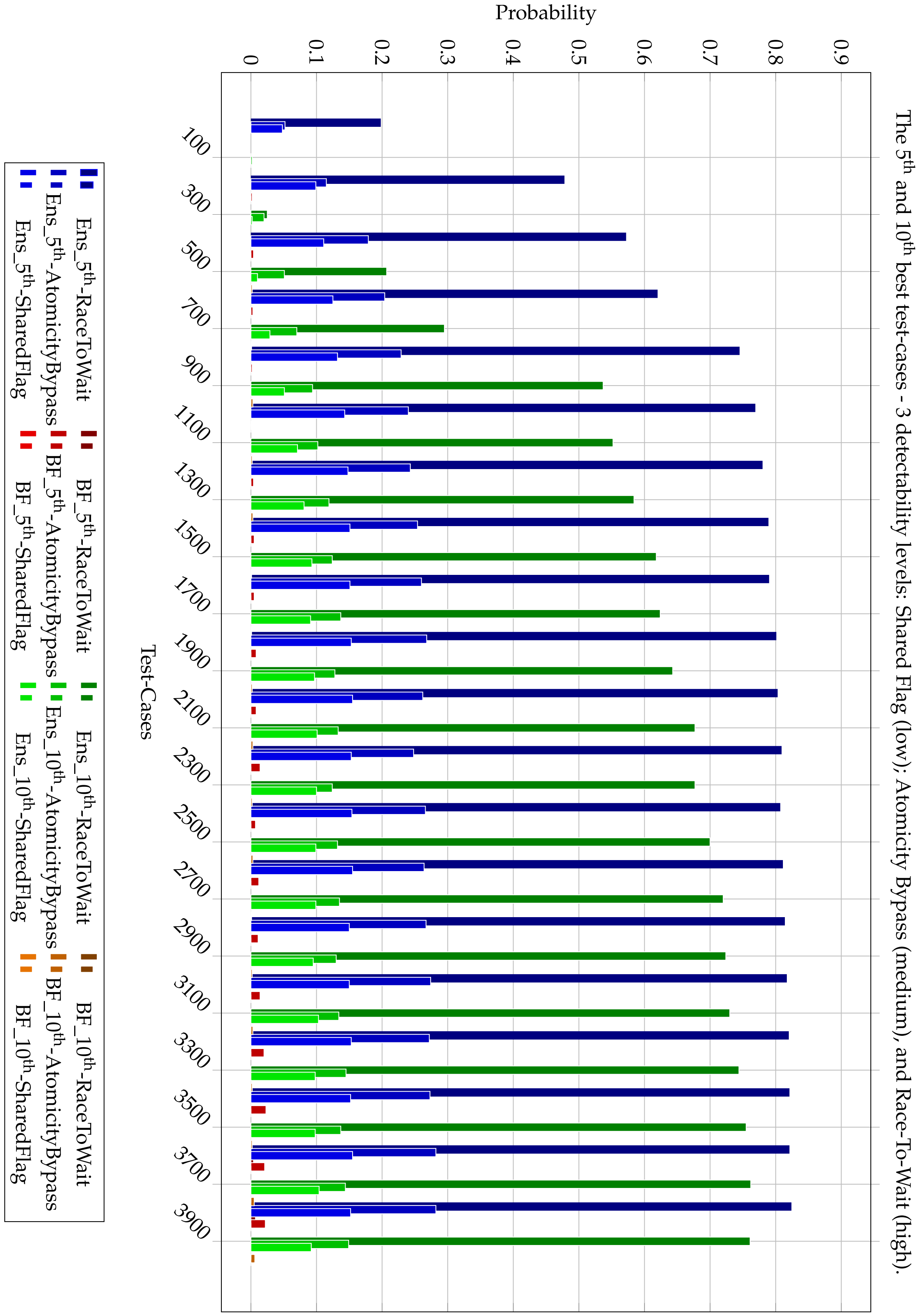

Figure 5. The 5th and 10th best test-cases probability. In three detectability levels of problems that cover three ranges of probability: Shared Flag (low); Atomicity Bypass (medium), and Race-To-Wait (high). Each bar is one experiment and based on 50 independent runs. The X axis is all 20 test-cases, and for each problem, 2 chosen methods (*Ens*&*BF*) and 3 (out of 17) problems are shown. The y-axis is the maximum probability for the best test-case.

We note that for most problems, especially those in the medium-to-high difficulty range (true probability of failure between 0.2 and 0.6), the classifier-based method (*Ens*) consistently outperforms all baselines. On average, the probability of detecting a fault in the top-1 test-case rises from 22% with *BF* to 45% with *Ens*, a relative improvement of more than 100%. In the top-10 ranking, the average success rate jumps from 40% (*BF*) to 72% (*Ens*), with 9 out of 10 problem instances showing a statistically significant advantage (Wilcoxon one-sided test, $p < 0.01$).

These graphs support the hypothesis that even partial ranking from learned models can significantly improve fault detection.

Averaging over the entire benchmark, the 5^{th} best test-case chosen by *Ens* triggers the bug 31% of the time, vs. 11% for *BF*; for the 10^{th} best candidate, the rates are 24% vs. 6% (both differences significant at $p < 0.001$).

7.4. Pairwise Statistical Significance Analysis

This section presents a detailed pairwise statistical comparison between the evaluated methods using one-sided Wilcoxon signed-rank tests, in accordance with contemporary best practices for nonparametric analysis [30]. Table 3 displays the results across all 17 benchmark problems, using the best-case test input identified for each method. Each row corresponds to a benchmark problem, and each column reports the outcome of a directional hypothesis comparing a pair of methods *Ens*, *BF*, *SA*, and *GA* where aech cell shows the p-value for the hypothesis that the method in the row significantly outperforms the method in the column.

Green cells indicate statistically significant superiority ($p < 0.05$), gray cells indicate nonsignificant differences ($p \geq 0.05$), and red cells represent reversed directions. In total, the table comprises 68 directional pairwise comparisons (17 problems × 4 method pairs). The *Ens* method shows a particularly strong trend: it significantly outperforms *BF* in 15 out of 17 cases, SA in all 17 cases, and *GA* in 16 cases. This consistency reflects both strong absolute performance and low variability. In contrast, *BF* significantly outperforms SA in 15 problems but offers a limited advantage over *GA*, outperforming it significantly in only two problems. *GA* and SA, on the other hand, do not significantly outperform any other method in any problem, indicating weaker and less consistent behavior.

Only 14 of the 102 comparisons are statistically inconclusive (gray cells), highlighting that the majority of results are directional and meaningful. This overall structure reveals a clear performance hierarchy: *Ens* consistently outperforms all others, *BF* performs reliably better than SA, while SA and *GA* rarely, if ever, demonstrate statistical superiority. These patterns underline the robustness of the *Ens* approach across diverse concurrency bug types and failure modes. The statistical evidence supports its adoption as a dominant strategy for test amplification in multithreaded programs.

Table 3. Wilcoxon one-sided signed-rank test results on **best** scores. Each cell shows the p-value for the hypothesis that the left method performs better than the right (e.g., Ans→BF). Green cells indicate significant results ($p \leq 0.05$), gray cells indicate no significance ($0.05 < p < 0.95$), and red cells indicate evidence in the opposite direction.

Problem	Ens→GA	Ens→BF	Ens→SA	GA→BF	GA→SA	BF→SA
AtomicityBypass	0.002	<0.001	<0.001	0.566	<0.001	<0.001
BrokenBarrier	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
BrokenPeterson	0.002	<0.001	<0.001	<0.001	<0.001	<0.001
DelayedWrite	0.003	<0.001	<0.001	<0.001	<0.001	<0.001
FlagedDeadlock	0.003	0.002	<0.001	<0.001	<0.001	<0.001
IfNotWhile	0.003	0.003	0.001	<0.001	<0.001	<0.001
LockOrderInversion	0.984	0.434	0.003	0.054	<0.001	<0.001
LostSignal	<0.001	<0.001	<0.001	0.174	<0.001	<0.001
PartialLock	0.295	0.214	0.003	0.130	<0.001	<0.001
PhantomPermit	0.003	0.003	0.003	0.127	0.003	0.011
RaceToWait	0.007	0.003	0.003	<0.001	<0.001	0.996
RacyIncrement	0.003	<0.001	<0.001	<0.001	<0.001	<0.001
SemaphoreLeak	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
SharedCounter	0.003	0.003	0.001	<0.001	<0.001	<0.001
SharedFlag	0.003	0.003	<0.001	<0.001	<0.001	<0.001
SignalThenWait	0.002	<0.001	<0.001	<0.001	<0.001	0.014
SleepingGuard	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

7.5. Convergence Analysis Across Methods

In this analysis, we study convergence patterns by plotting the probability of success as a function of the number of test-cases, aggregated over all 17 benchmark problems. Figure 2 provides a bird's-eye view that captures performance trends at four representative budget levels: 500, 1100, 2100, and 3900 test-cases. For each method, *Ens*, *BF*, *SA*, and *GA*, we plot the best failure-inducing probability obtained per problem, averaged over 50 independent runs.

The ensemble classifier-based method (*Ens*) exhibits remarkably fast and stable convergence. At just 500 test-cases, *Ens* already achieves a mean success probability of 51.8% across all problems. This value rises to 56.4% at 1100, 58.7% at 2100, and reaches 59.8% at 3900. These gains are not only large in absolute terms but also consistently achieved across a diverse range of problem types. This demonstrates the model's ability to generalize its learned prioritization across different failure patterns.

In contrast, the brute-force approach (*BF*) converges slowly. It begins with a mean success rate of just 3.1% at 500 test-cases, improving modestly to 6.2% at 1100, 10.0% at 2100, and only 13.6% at 3900 test-cases. This linear and limited improvement confirms the inefficiency of uninformed random exploration.

Simulated Annealing (*SA*) and Genetic Algorithm (*GA*) fall between these extremes. *SA* improves from 1.5% (at 500 test-cases) to 3.9% (at 3900), with substantial stagnation between checkpoints, reflecting a limited capacity to escape local minima. *GA* achieves higher starting performance at 500 test-cases (mean 8.1%) and improves more rapidly than *SA*, reaching 17.3% at 3900, but still falls far short of *Ens*.

Overall, these convergence patterns reinforce the strength of learning-guided strategies. *Ens* not only achieves the highest final probabilities but also reaches them faster, demonstrating both sample efficiency and consistent generalization. This advantage is particularly valuable in real-world testing scenarios where test execution budgets are constrained and high-probability failure discovery is critical.

7.6. Summary of Key Findings

Our evaluation of four amplification methods, Brute-Force (*BF*), Simulated Annealing (*SA*), Genetic Algorithm (*GA*), and Ensemble Classification (*Ens*), across 17 benchmark concurrency problems, led to several key findings that integrate both method-specific behavior and cross-cutting insights:

Learning-based amplification significantly outperforms uninformed approaches. The ensemble classifier (*Ens*) consistently achieved the highest bug-triggering probabilities across nearly all test-case budgets and problems. With just 500 test-cases, *Ens* reached average success probabilities exceeding 0.53, whereas *BF*, *SA*, and *GA* remained below 0.13. At the full budget of 3900 test-cases, *Ens* achieved near-perfect detection (over 0.9 probability) in more than half of the problems, including *LockOrderInversion*, *SignalThenWait*, and *IfNotWhile*.

***Ens* converges faster and with fewer test-cases.** While *BF*, *SA*, and *GA* showed gradual or erratic improvements, *Ens* rapidly identified failure-inducing cases. For example, in *RacyIncrement*, *Ens* surpassed 0.9 success probability with fewer than 1100 test-cases, while *GA* plateaued at 0.07 and *BF* at 0.03 even after 3900 cases. This sample efficiency makes *Ens* especially valuable for real-world systems with costly or time-limited testing resources.

Traditional search methods offer limited scalability. *BF* showed minimal improvement over increasing test budgets, with average performance rarely exceeding 0.15 across problems. *SA*'s performance improved modestly but remained inconsistent, failing to trigger bugs in several hard problems like *SharedFlag* and *SemaphoreLeak*. *GA* was more effective than *BF* and *SA* in moderately complex problems but still lagged behind *Ens* in both speed and final success rates.

Problem hardness varies significantly and affects method effectiveness. Some problems were consistently easy (e.g., *SignalThenWait* and *LockOrderInversion*) and triggered by all methods to varying degrees. Others, such as *SharedFlag*, *SemaphoreLeak*, and *BrokenBarrier*, remained elusive, with only *Ens* achieving meaningful success (e.g., 0.49 in *SemaphoreLeak* vs. <0.03 for others). This suggests that learning-based methods are better suited for navigating complex or deceptive search spaces.

***Ens* robustness is evident across all tested budgets.** The bird's-eye view (Figure 2) shows that across all 17 problems and at every tested budget (500, 1100, 2100, and 3900), *Ens* consistently led or tied for the highest success rate. Notably, in 13 out of 17 problems, *Ens* reached probabilities above 0.85 with 3900 test-cases, while *GA* exceeded 0.5 in only 7, *SA* in two, and *BF* in one.

Integration of feedback powers *Ens* performance. Unlike the other methods, which rely on sampling or mutation heuristics, *Ens* uses supervised learning to predict and prioritize high-risk inputs. This allows it to generalize from early failures, focusing search efforts efficiently. The result is not only higher probabilities of detecting bugs but also significantly fewer wasted executions.

Ablation Study. We conducted ablation studies by removing components from the ensemble classifier and modifying its sampling heuristics. Specifically, we evaluated simplified variants of our pipeline, such as omitting SMOTE or disabling passthrough to the meta-learner. These reduced versions consistently underperformed relative to the full classifier configuration we present in the paper. In several cases, the simplified ensemble-based methods even performed worse than the brute-force baseline, highlighting the importance of each pipeline component in achieving effective bug amplification.

Our findings support the superiority of learning-guided search for amplifying concurrency bugs. *Ens* is not only more effective in absolute terms but also more efficient, scalable, and robust across problem domains and budgets. These characteristics make it a promising default choice for future automated bug-amplification tools.

8. Related Work

8.1. Concurrency Bug Debugging Methods

Over the past ten years, concurrent systems bug hunting has evolved significantly, driven by the growing complexity of multithreaded software and the critical need to detect concurrency bugs, such as data races, deadlocks, and atomicity violations.

A survey of academic papers from sources like IEEE Xplore, ACM Digital Library, and SpringerLink reveals three dominant methodological categories: static analysis, dynamic analysis, and model checking, each encompassing diverse techniques with unique trade-offs, industrial applications, and ongoing refinements.

Static analysis: Techniques that scrutinize code without execution include abstract interpretation, data-flow analysis, type systems, symbolic execution, and machine learning-based bug prediction. Abstract interpretation [31] models program semantics to detect bugs across all paths, offering early detection but often producing false-positives due to over-approximation. Data-flow analysis [32] tracks dependencies and works well in structured parallelism (e.g., OpenMP), though its generalization to unstructured concurrency remains limited. Type systems, such as Rust's ownership model [33], prevent bugs at compile-time with minimal runtime cost, though they require full language adoption. Symbolic execution [34] can uncover deep concurrency bugs through path exploration but suffers from path explosion. Machine learning approaches [35] learn patterns from code to predict concurrency bugs but depend heavily on the availability of labeled data. Tools like Coverity leverage static analysis in the industry, though concurrency-specific precision remains a challenge.

Dynamic analysis: This category executes programs to observe runtime-behavior and includes methods like thread-aware fuzzing, runtime monitoring, and record-and-replay. Thread-aware fuzzing [36] explores interleavings to expose real bugs but may suffer from incomplete coverage. Runtime monitoring [37] provides precise race detection at the cost of performance overhead. Record-and-replay [38] facilitates debugging by reproducing execution paths, albeit with recording overhead. Tools like ThreadSanitizer are widely used due to their balance of effectiveness and performance.

Model checking: This technique provides formal verification by exhaustively exploring program state-spaces. Explicit-state model checking [39] can prove correctness but is vulnerable to state explosion. Bounded model checking [40] uses SAT/SMT solvers to explore execution within depth bounds, trading completeness for scalability. Abstraction-based techniques [41] simplify systems but risk imprecision. Compositional approaches [42] decompose systems for modular checking, though assumptions can break down. Statistical model checking [43] approximates correctness via sampling and is used in domains like embedded systems and aerospace, where formal guarantees are difficult to obtain.

Hybrid approaches have emerged to balance strengths and weaknesses, e.g., KRACE [44] employs thread-scheduling perturbation and fuzzing to detect data races in kernel file systems. Benchmarks such as the Linux kernel and SPEC CPU continue to reveal challenges: static methods must reduce false-positives, dynamic tools need improved coverage, and model checking must scale better. Future directions involve tighter integration of these methods and greater automation to support concurrency bug detection at scale.

8.2. Concurrency Bug Datasets

The study of concurrency bugs has led to the development of a wide range of datasets, each designed to capture specific aspects of concurrent programming behavior. These datasets can be grouped into four broad categories: general-purpose concurrency bug datasets, language-specific datasets, smart contract datasets, and fuzzing-based datasets. Below, we summarize key datasets from each category, highlighting their structure, scope, and contributions to academic research.

General-Purpose Concurrency Bug Datasets: Early work in concurrency bug research focused on real-world software systems. [45] compiled 105 concurrency bugs from widely used applications such as MySQL and Apache. The dataset revealed common bug patterns and has influenced numerous studies in static and dynamic analysis. CHESS [46], developed by Microsoft Research, explores all thread interleaving to find concurrency bugs. RACEBENCH [47] is a benchmark suite containing 29 multithreaded programs with known races, offering a standardized environment for testing race detectors. DETECT [48] uses dynamic analysis and communication graphs to identify concurrency bugs.

Language-Specific Datasets: With the growing demand for language-aware tools, several datasets were created targeting Java and Go. For Java, JaConTeBe [49] includes 47 confirmed bugs from 8 Java projects. Defects4J [50] is a curated repository of real-world Java bugs, used extensively in software testing and repair. Bears [51] collects bugs from CI pipelines to support automated program repair. ManySStuBs4J [52] offers over 500k single-statement bugs, indirectly supporting concurrency research. For Go, the Go Concurrency Bug Collection [53] contains 171 bugs from six Go applications [3]. GoBench [54] expands this effort with 82 real bugs and 103 bug kernels.

Smart Contract Datasets: With the rise of blockchain applications, concurrency issues in smart contracts gained prominence. ConFuzzius [55] combines evolutionary fuzzing and symbolic execution to detect concurrency-related bugs in Ethereum smart contracts, building a dataset of known vulnerabilities.

Fuzzing-Based Datasets: Grey-box fuzzing has proven valuable for stress-testing concurrent applications. MUZZ [36] presents a thread-aware fuzzing method for multithreaded programs, featuring a dataset of real-world apps annotated with concurrency bugs.

These datasets continue to support advances in concurrency research, enabling reproducibility, benchmarking, and tool evaluation across diverse programming environments.

9. Detailed Description of the Benchmark Problems

This section details the benchmark problems. For every problem, we document (i) the scenario, (ii) its observable effect, (iii) the underlying root cause, and (iv) a concise insight:

9.1. Atomicity Bypass: Unexpected Data from Lock Misuse

Description: Simulates two threads updating a shared counter under the false assumption that a critical section is properly protected. Each thread acquires a mutex, reads the counter, but then mistakenly releases the mutex before performing the update. As a result, both threads read the same value (e.g., 0), and both write back 1, overwriting each other's increment. The final result is data corruption: the counter appears to have only been incremented once.

Effect: A clearly unexpected data outcome, where both threads read the same initial value of the counter and write back identical updates, resulting in a lost increment. This leads to data corruption, as the counter reflects only one update instead of two, violating correctness expectations.

Root Cause: Misuse of concurrency primitives: The locking discipline was violated by releasing the mutex too early.

Insight: This demonstrates that simply using synchronization tools is insufficient - they must be used correctly and consistently to protect shared operations.

Pseudo Code:

```
Thread 0
while mutex == 1 do
    wait()
end while
mutex ← 1
local ← counter
mutex ← 0          ▷ BUG: unlock before update
counter ← local + 1
```

```
Thread 1
while mutex == 1 do
    wait()
end while
mutex ← 1
local ← counter
mutex ← 0
counter ← local + 1
```

[3] [https://github.com/system-pclub/go-concurrency-bugs

9.2. Broken Barrier: Deadlock from Barrier Misuse with Incorrect Participant Count

Description: Three threads increment a shared variable and call `SignalAndWait()` on a barrier that is configured for only two participants. One thread calls `SignalAndWait()` twice before resetting the barrier, violating the expected usage pattern.

Effect: This misconfiguration can lead to deadlock, as some threads may wait indefinitely for signals that never arrive. It may also cause assertion failures if the synchronization logic assumes a specific number of participants.

Root Cause: A misuse of primitives, where the barrier is used in a way that contradicts its intended configuration.

Insight: This problem illustrates the importance of synchronization primitives being correctly configured for the actual number of participating threads. Misuse of barriers can lead to subtle and difficult-to-diagnose concurrency failures.

Pseudo Code:

```
Thread 0
while true do
    Increment(ref fireballCharge)
    barrier.SignalAndWait()
    if fireballCharge < 2 then
        Debug.Assert(false)
    end if
    fireball()
end while
```

```
Thread 1
while true do
    Increment(ref fireballCharge)
    barrier.SignalAndWait()
end while
```

```
Thread 2
while true do
    Increment(ref fireballCharge)
    barrier.SignalAndWait()
    barrier.SignalAndWait()
    fireballCharge ← 0            ▷ BUG: reset can occur too early
end while
```

9.3. Broken Peterson: Mutual Exclusion Violation in Generalized Peterson's Algorithm

Description: This problem involves a generalized version of Peterson's algorithm for four processes. The implementation uses arrays to track process levels and a `last_to_enter` array to manage entry ordering. However, a critical assignment to `last_to_enter[level]` is omitted, breaking the algorithm's tie-breaking logic.

Effect: Multiple processes may enter the critical section concurrently, leading to a concurrent access.

Root Cause: A missing or weak guard in the synchronization protocol, specifically, a missing update in the entry coordination mechanism.

Insight: This example highlights how even small implementation errors in well-established algorithms can undermine their correctness. It underscores the need for rigorous validation of synchronization logic, especially in generalized or modified versions of classic algorithms.

Pseudo Code:

General Peterson Algorithm (Process i)

```
while true do
    for $\ell = i$ to $n - 2$ do
        last_to_enter[$\ell$] $\leftarrow i$                    ▷ Bug: wrong order
        levels[$i$] $\leftarrow \ell$
        while exists $k \neq i$ such that levels[$k$] $\geq \ell$ and last_to_enter[$\ell$] = $i$ do
            wait
        end while
    end for
    critical_section()
    levels[$i$] $\leftarrow -1$
    remainder_section()
end while
```

9.4. Delayed Write – Assertion Failure from Non-Atomic Test-and-Set Simulation

Description: A simulation models a `test-and-set` operation where one thread sets a shared variable `x` to a target value. However, another thread may interleave and modify `x` during a context switch, violating the assumption that `x` remains unchanged after being set.

Effect: A concurrent access and unexpected data, often manifesting as an assertion failure when the invariant `x == target` is violated.

Root Cause: An incorrect command ordering stemming from the test-and-set logic. The thread reads and later writes to `x`, but a context switch between these steps allows another thread to intervene and modify the variable, violating expected execution order.

Insight: This case illustrates how concurrency bugs can emerge even in simulated atomic operations if the underlying memory operations are not properly synchronized. It emphasizes the importance of true atomicity in synchronization primitives.

Pseudo Code:

Thread 0

```
global x
x=TARGET
if x != TARGET:
    assert (x!=TARGET)
```

Thread 1

```
global x
x = 3
```

9.5. Flagged Deadlock: Deadlock Risk from Complex Locking

Description: Involves two threads using a combination of locking strategies, including recursive locks, try-locks, and conditional logic based on shared flags. The complexity of the locking protocol introduces multiple paths for acquiring locks, some of which may conflict or fail to release locks properly.

Effect: A heightened risk of deadlock, as threads may become stuck waiting for locks that are never released or acquired in inconsistent orders.

Root Cause: A combination of misuse of primitives and non-cooperative scheduling, exacerbated by the use of active waiting (spin locks) instead of blocking synchronization.

Insight: This case highlights the dangers of over-engineering synchronization logic. Complex locking schemes, especially those involving conditional paths and re-entrant locks, are prone to subtle bugs and should be avoided in favor of simpler, more predictable designs.

Pseudo Code:

```
Thread 0
while true do
    if Monitor.TryEnter(mutex) then
        Monitor.Enter(mutex3)
        Monitor.Enter(mutex)
        critical_section()
        Monitor.Exit(mutex)
        Monitor.Enter(mutex2)
        flag ← false
        Monitor.Exit(mutex2)
        Monitor.Exit(mutex3)
    else
        Monitor.Enter(mutex2)
        flag ← true
        Monitor.Exit(mutex2)
    end if
end while
```

```
Thread 1
while true do
    if flag then
        Monitor.Enter(mutex2)
        Monitor.Enter(mutex)          ▷ BUG: mutex is held
        flag ← false
        critical_section()
        Monitor.Exit(mutex)
        Monitor.Enter(mutex2)         ▷ BUG:already held it
    else
        Monitor.Enter(mutex)
        flag ← false
        Monitor.Exit(mutex)
    end if
end while
```

9.6. If-Not-While: Deadlock and Missed Signals from Condition Variable Misuse

Description: Two consumer threads wait on a shared queue using `Monitor.Wait(mutex)` when the queue is empty. A producer thread enqueues data and signals all waiting consumers using `Monitor.PulseAll(mutex)`. However, the consumers guard the wait with an `if` statement rather than a `while` loop, failing to re-check the condition upon waking.

Effect: This leads to two possible effects: deadlock, if a consumer misses a signal and waits indefinitely, or unexpected data loss, if a consumer proceeds without the queue being properly populated.

Root Cause: A race condition caused by a weak guard; the failure to revalidate the condition after waking allows incorrect assumptions about the system state.

Insight: This problem reinforces the importance of using guarded waits with `while` loops when working with condition variables, ensuring that threads only proceed when the condition they depend on is truly satisfied.

Pseudo Code:

```
Thread 0
while true do
    Monitor.Enter(mutex)
    if queue.Count == 0 then
        Monitor.Wait(mutex)          ▷ release & wait
    end if
    queue.Dequeue()
    Monitor.Exit(mutex)
end while
```

```
Thread 1
while true do
    Monitor.Enter(mutex)
    if queue.Count == 0 then
        Monitor.Wait(mutex)          ▷ release & wait
    end if
    queue.Dequeue()
    Monitor.Exit(mutex)
end while
```

```
Thread 2
while true do
    Monitor.Enter(mutex)
    queue.Enqueue(42)
    Monitor.PulseAll(mutex)
    Monitor.Exit(mutex)
end while
```

9.7. Lock Order Inversion: Deadlock from Inconsistent Lock Acquisition Order

Description: In this classic concurrency scenario, two threads attempt to acquire two shared locks but do so in opposite orders. Thread 0 first locks `mutex1` and then attempts to acquire `mutex2`, while Thread 1 begins by locking `mutex2` and then proceeds to request `mutex1`. This inversion in lock acquisition order creates a circular wait condition: each thread holds one lock and waits indefinitely for the other to release the second, which never happens.

Effect: A deadlock, where both threads are permanently blocked, unable to make progress.

Root Cause: An incorrect order, a well-known concurrency design flaw where multiple threads acquire shared resources in inconsistent sequences. When such errors occur, they can easily lead to circular dependencies, especially in systems that lack a global lock acquisition policy.

Insight: This problem exemplifies the dangers of uncoordinated locking strategies in multithreaded environments. It highlights the importance of enforcing a consistent global order for acquiring multiple locks, a practice that can prevent deadlocks and ensure system liveness. The scenario is a textbook case of "lock inversion", a term often used to describe such deadlock-prone patterns in concurrent programming.

Pseudo Code:

```
Thread 0
Monitor.Enter(mutex1);
Monitor.Enter(mutex2);
critical_section();
Monitor.Exit(mutex1);
Monitor.Exit(mutex2);
```

```
Thread 1
Monitor.Enter(mutex2);
Monitor.Enter(mutex1);
critical_section();
Monitor.Exit(mutex2);
Monitor.Exit(mutex1);
```

9.8. Lost Signal: Deadlock from Missed Signal in Condition Variable Coordination

Description: Two threads coordinate using a shared condition variable. Thread 0 waits for a flag to become true using an `if` statement and then calls `wait()`. Thread 1 sets the flag and sends a notification using `notify_all()`. If Thread 1 sends the signal before Thread 0 begins waiting, the signal is lost, and Thread 0 waits indefinitely.

Effect: A deadlock, as Thread 0 never receives the signal it depends on.

Root Cause: A weak guard: Thread 0 fails to re-check the condition after waking and uses an `if` statement instead of a `while` loop to guard the wait.

Insight: This problem reinforces a key principle in concurrent programming: condition variables must be used with guarded waits that revalidate the condition upon waking. This ensures correctness even in the presence of spurious wakeups or early notifications.

Pseudo Code:

Thread 0 (Waiter - Weak Guard)

```
lock(mutex)
if flag == false then
    cv.wait(mutex)    ▷ Bug: only checks once
end if
proceed_assuming_flag_true()
unlock(mutex)
```

Thread 1 (Signaler)

```
lock(mutex)
flag ← true
cv.notify_all()
unlock(mutex)
```

9.9. Partial Lock: Race Condition from Insufficient Lock Coverage

Description: Two threads manipulate a shared variable `i` under a locking mechanism. Thread 0 increments `i` by 2 and checks whether `i == 5`, while Thread 1 decrements `i` by 1. Although both threads use a lock, the locking does not encompass all relevant operations or ensure proper coordination between them. As a result, the interleaving of operations can lead to unexpected values of `i`, potentially triggering assertion failures.

Effect: An unexpected data or incorrect computation, as the shared state evolves in ways not anticipated by the program logic.

Root Cause: A missing or weak guard due to the lock is not applied consistently across all accesses and updates to the shared variable, allowing unsafe interleaving.

Insight: This example illustrates that merely using locks is not enough; they must be applied comprehensively and consistently to protect all shared state interactions.

Pseudo Code:

Thread 0

```
while true do
    Monitor.Enter(mutex)
    $i \leftarrow i + 2$
    critical_section()
    if $i = 5$ then
        Debug.Assert(false)    ▷ BUG: This assert can fail
    end if
    Monitor.Exit(mutex)
end while
```

Thread 1

```
while true do
    Monitor.Enter(mutex)
    $i \leftarrow i - 1$
    critical_section()
    Monitor.Exit(mutex)
end while
```

9.10. Phantom Permit: Mutual Exclusion Violation from Semaphore Misuse

Description: Two threads share a binary semaphore intended to serialise entry to a critical section. Thread 0 performs the canonical `Wait`–critical section–`Release` sequence, preserving mutual exclusion. Thread 1, by contrast, invokes `Wait(timeout)`. If the timeout expires, it nevertheless executes `Release`, effectively inserting an extra permit into the semaphore (a "phantom" permit).

Effect: Concurrent access arises when the phantom permit allows both threads to enter the critical section simultaneously, enabling interleaved operations that can corrupt shared state or violate higher-level invariants.

Root Cause: The defect is rooted in a misuse of concurrency primitives: issuing `Release` without first holding the semaphore breaks the required one-to-one pairing of `Wait`/`Release`. This increases the semaphore's count spuriously and defeats its mutual-exclusion guarantee.

Insight: Correct semaphore protocols demand that every `Release` correspond to a successful `Wait`. Introducing time-limited waits without compensating logic must be done carefully; otherwise, phantom permits can emerge and silently undermine critical-section protection.

Pseudo Code:

```
Thread 0 (Acquirer)
while semaphore == 0 do
    wait()
end while
semaphore -= 1
critical_section()
semaphore += 1
```

```
Thread 1 (Timed Failer)
if timeout then
    /* never acquired semaphore */
    semaphore += 1            ▷ BUG: false release
end if
```

9.11. Race-To-Wait: Deadlock from Non-Atomic Coordination

Description: Two threads attempt to synchronize based on a shared counter `waiters`. Each thread increments the counter and then waits for it to reach a specific value (e.g., 2) before proceeding. However, the increment and check operations on `waiters` are not atomic. Both threads may read the value 1 simultaneously before either has incremented it again, leading them both to wait forever for the counter to reach 2, which never happens.

Effect: A classic deadlock, even though no explicit locking mechanism is involved.

Root Cause: A non-atomic operation on shared state: the threads make decisions based on stale or incomplete views of shared memory.

Insight: This example highlights how even minimalistic, lock-free coordination can result in liveness failures if atomicity is not respected.

Pseudo Code:

```
Thread 0
temp ← waiters
waiters ← temp + 1    ▷ BUG: non-atomic
if waiters < 2 then
    wait()
end if
```

```
Thread 1
temp ← waiters
waiters ← temp + 1        ▷ Same bug
if waiters < 2 then
    wait()
end if
```

9.12. Racy Increment: Race Condition from Non-Atomic Compound Operations

Description: This problem illustrates a subtle but critical flaw in assuming that compound operations are atomic. Two threads execute the expression `a = a + 1; if (a == 1) enter critical section`, intending to allow only the first thread that increments `a` to 1 to enter the critical section. However, this logic fails under concurrent execution because the operation `a = a + 1` is not atomic-it decomposes into a sequence of read, increment, and write steps. If both threads interleave during these steps, they may each observe `a` as 0, increment it to 1, and both proceed into the critical section.

Effect: A concurrent access, where both threads enter a region that was intended to be accessed by only one. This leads to unexpected data, as the shared state is manipulated under the false assumption of exclusivity.

Root Cause: A non-atomic operation stemming from the non-atomicity of the increment-and-check sequence. Without synchronization, the interleaving of operations allows both threads to satisfy the condition `a == 1` simultaneously.

Insight: This example underscores the importance of using atomic operations or explicit synchronization mechanisms, such as locks or atomic primitives, when accessing shared variables. It also highlights how deceptively simple code can harbor concurrency bugs if the underlying memory operations are not properly understood.

Pseudo Code:

```
Thread 0
temp ← $a$
temp ← temp +1
$a$ ← temp ▷BUG: non-atomic update may interleave
if $a = 1$ then
    critical_section()
end if
```

```
Thread 1 (Expanded Assignment)
temp ← $a$
temp ← temp +1
$a$ ← temp            ▷ BUG: same
if $a = 1$ then
    critical_section()
end if
```

9.13. Semaphore Leak: Mutual Exclusion Violation from Semaphore Misuse

Description: Involves two threads using a semaphore to control access to a critical section. Thread 0 follows the standard `Wait`–critical section–`Release` pattern. Thread 1, however, performs a time-limited `Wait` and calls `Release` regardless of whether it successfully acquired the semaphore.

Effect: This behavior can corrupt the semaphore's internal count, allowing multiple threads to enter the critical section simultaneously, a clear concurrent access.

Root Cause: A misuse of primitive: releasing a semaphore without a corresponding acquisition violates the expected one-to-one pairing of `Wait` and `Release`.

Insight: This example underscores the importance of maintaining strict discipline when using semaphores. Any deviation from the expected protocol can compromise the integrity of the synchronization mechanism.

Pseudo Code:

```
Thread 0
while true do
    semaphore.Wait()
    critical_section()
    semaphore.Release()
end while
```

```
Thread 1
while true do
    if semaphore.Wait(500) then        ▷ Wait with timeout
        critical_section()
        semaphore.Release()
    else
        semaphore.Release() ▷ BUG: release without own
    end if
end while
```

9.14. Shared Counter: Mutual Exclusion Violation from Unsynchronized Counter

Description: Involves two threads incrementing a shared counter and entering a critical section based on different thresholds, one at a count of 5, the other at 3. The counter is not protected by any synchronization mechanism, allowing updates to interleave unpredictably.

Effect: Both threads may enter the critical section simultaneously or at unintended times, leading to a concurrent access and unexpected data.

Root Cause: A race condition due to the non-atomic operation and check of the shared counter.

Insight: This example demonstrates the necessity of synchronizing access to shared counters, especially when control flow decisions depend on their values. Without atomicity, even simple arithmetic can lead to concurrency failures.

Pseudo Code:

Five-Headed Dragon

```
while true do
    counter $\leftarrow$ counter $+1$
    if counter == 5 then
        critical_section()
    end if
end while
```

Three-Headed Dragon

```
while true do
    counter $\leftarrow$ counter $+1$
    if counter == 3 then
        critical_section()
    end if
end while
```

9.15. Shared Flag: Mutual Exclusion Violation from Weak Boolean Flag Guard

Description: Demonstrates the inadequacy of using a simple Boolean flag to enforce mutual exclusion. Two threads share a flag and use it to guard a critical section. Each thread spins while the flag is `true`, sets it to `true`, enters the critical section, and then resets it to `false`. However, the check (`flag != false`) and the update (`flag = true`) are not atomic. If one thread is preempted after checking the flag but before setting it, the other thread may also pass the check and set the flag, resulting in both threads entering the critical section concurrently.

Effect: A concurrent access, where the critical section is accessed simultaneously by multiple threads, leading to potential data corruption or logic errors.

Root Cause: A weak guard-the synchronization mechanism fails to ensure atomicity between the check and the update. This highlights the need for atomic test-and-set operations or proper locking mechanisms to enforce exclusive access.

Insight: This highlights the need for atomic test-and-set operations or proper locking mechanisms to enforce exclusive access.

Pseudo Code:

First Army

```
while true do
    while flag $\neq$ false do
        /* busy wait */
    end while
    flag $\leftarrow$ true
    critical_section()
    flag $\leftarrow$ false
end while
```

Second Army

```
while true do
    while flag $\neq$ false do
        /* busy wait */
    end while
    flag $\leftarrow$ true    ▷ BUG: both can pass check
    critical_section()
    flag $\leftarrow$ false
end while
```

9.16. Signal-Then-Wait – Deadlock from Premature Signaling in Condition synchronization

Description: Two threads coordinate using a shared flag and condition variable. The signaling thread sets the flag and calls `notify_all()` before the waiting thread has entered the blocking wait. Although the waiting thread uses a correct `while` guard around the condition variable, the notification is missed entirely because the thread was not waiting yet.

Effect: A clear deadlock: the waiting thread blocks indefinitely, even though the condition it depends on was fulfilled. This occurs because condition variable signals do not persist - if a signal is sent before a thread is waiting, it is lost.

Root Cause: An incorrect ordering of commands - the signal is issued before the synchronization context is established. This leads to a fundamental timing mismatch between threads.

Insight: This pattern highlights that the timing of signal delivery in condition variable synchronization is critical. Signals must occur only after the corresponding wait condition has been armed, or the system risks falling into liveness failures such as deadlock.

Pseudo Code:

doi:10.20944/preprints202507.2304.v1

```
Thread 0 (Waiter)
lock(mutex)
while flag == false do
    wait_blocked ← true
    wait(cv, mutex)  ▷ BUG: signal already sent
end while
use_resource()
unlock(mutex)
```

```
Thread 1 (Signaler)
flag ← true          ▷ BUG: condition updated before wait begins
lock(mutex)
notify_all(cv)
unlock(mutex)
```

9.17. Sleeping Guard: Deadlock from Missing or Weak Guard

Description: Presents a subtle but powerful failure in the use of condition synchronization. A consumer thread checks a queue, and if it's empty, sets a `waiting` flag and waits. A producer thread checks for the flag and enqueues data. The issue occurs if the producer enqueues a new item before the consumer sets the flag; the consumer misses the notification and remains blocked indefinitely.

Effect: A classic deadlock, in which the consumer thread remains permanently blocked waiting for a signal that was sent before it armed the condition, while the producer continues indefinitely, leaving the system with no forward progress.

Root Cause: A missing or weak guard: the consumer waits based solely on a flag without rechecking the real shared resource (the queue). In such designs, the wait must be governed by a guard that accurately reflects the synchronization invariant, and it must be re-evaluated after any wake-up event.

Insight: Without such a recheck, typically enforced with a `while` loop, the thread risks sleeping forever, even though the condition it depends on has already been satisfied.

Pseudo Code:

```
Consumer
if queue.empty() then
    waiting ← true
    sleep()                  ▷ BUG: doesn't recheck Q
end if
item ← queue.pop()
process(item)
```

```
Producer
queue.push(item)
if waiting then
    waiting ← false
end if
```

10. Limitations of the Proposed Approach

While our approach to bug amplification demonstrates strong empirical performance across a diverse set of concurrency problems, it is important to acknowledge its current limitations and boundaries of applicability.

Dependence on Parameter Sensitivity. Our method assumes that the probability of bug manifestation is meaningfully influenced by the input parameters exposed to the test generation engine. For systems where concurrency faults are insensitive to external parameters (e.g., bugs that manifest only due to internal scheduler decisions or deep state interactions), our black-box approach may offer limited leverage.

Curse of Dimensionality. As the dimensionality of the input space increases, learning an accurate regression model under a fixed testing budget becomes increasingly difficult. While our ensemble classifier demonstrated strong generalization in the studied benchmarks, its performance may degrade in higher-dimensional or sparsely populated input spaces, particularly when failure-inducing regions are extremely narrow.

Noise Sensitivity and Stochastic Feedback. Although we mitigate stochasticity through repeated executions, our framework is still subject to noise in failure observations. In scenarios where bug triggering is both rare and erratic, the resulting label noise can impair the quality of the learned models. This sensitivity places limits on how well regression-based methods can capture the underlying failure structure, especially early in the learning process.

Model Retraining Overhead. The iterative nature of our learning-based method requires frequent retraining of the classifier during test generation. While not a bottleneck in our Python-based implementation, this could become a concern for large-scale systems or industrial deployments with tight performance constraints, especially when test executions are costly.

No Schedule Control. Unlike techniques such as systematic concurrency testing or randomized schedulers, our approach does not manipulate the thread scheduler or execution order. As a result, bugs that require specific interleavings to manifest may remain elusive unless those conditions can be indirectly induced through parameter variation.

Despite these limitations, our method provides a practical, non-invasive tool for increasing the likelihood of bug detection in concurrent systems. It complements existing techniques by offering a black-box, input-driven strategy that is easy to integrate and effective across a wide range of problem types.

11. Summary and Conclusions

This paper addresses a fundamental challenge in software testing: reliably detecting concurrency bugs that manifest under rare interleavings and elusive execution schedules. These failures, though often critical, are notoriously hard to reproduce. To tackle this, we propose a probabilistic reformulation of the test generation task, treating bug detection as a problem of searching for inputs with maximized failure probability. This shift enables both a principled evaluation of search heuristics and the design of more effective testing strategies.

To evaluate our approach, we introduced a carefully curated benchmark of 17 multithreaded programs, each exhibiting a different concurrency failure. These programs span diverse root causes and error types, and the benchmark was built to ensure broad coverage and realism. For each problem, we examined the effectiveness of four black-box test generation methods: brute-force (*BF*), genetic algorithm (*GA*), simulated annealing (*SA*), and an ensemble classifier (*Ens*).Each method was executed in 50 independent trials per problem, producing a robust dataset for statistical comparison.

The central contribution of the paper lies in the design and implementation of the ensemble-guided test generation strategy. By treating bug-finding as a classification problem over test inputs, our method learns from past failures and adaptively focuses the search on high-potential areas of the input space. This method is fully black-box and does not require access to program internals. Our results demonstrate that this learning-based strategy consistently outperforms traditional heuristics across nearly all benchmark problems. Notably, *Ens* achieves higher detection rates using fewer test executions and converges more quickly to effective test inputs.

We further introduced a set of four graph-based analysis techniques that offer a detailed view of the methods' behavior: per-problem success curves, comparisons of top-ranked test-cases, convergence dynamics, and statistical significance heatmaps. These visual tools enabled us to examine method effectiveness from multiple perspectives and to identify patterns in both algorithmic performance and problem hardness. The analysis reveals that *Ens* not only provides early bug discovery but also maintains its advantage as the test budget increases, exhibiting both statistical robustness and practical scalability.

Finally, we propose a novel simulation-based search heuristic for continuous input spaces inspired by simulated annealing but guided by probabilistic failure gradients. This formulation opens avenues for future work in guided bug-amplification over high-dimensional input domains.

In conclusion, this paper contributes a new methodological framework for adaptive bug-amplification, introduces a reusable benchmark of concurrency problems, and provides compelling empirical evidence that ensemble-guided testing can substantially improve the reliability and efficiency of concurrency bug detection. We believe these findings advance the state-of-the-art in automated software testing and lay a foundation for broader adoption of machine-learning methods in fault localization and test generation.

Author Contributions: Conceptualization, all = (Y.W., G.W., O.M., E.F., G.A. & A.E.); methodology, all; software, Y.W.; validation, Y.W; formal analysis, all; investigation, Y.W.; resources, all; data curation, Y.W.; writing—original draft preparation, Y.W.; writing—review and editing, all; visualization, Y.W.; supervision, G.W.; project administration, all; funding acquisition, all. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by the Lynne and William Frankel Center for Computer Science and the Israeli Science Foundation grant No. 2714/19.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: We implemented a modular Python-based framework for conducting the amplification experiments The original data presented in the study are openly available in https://github.com/geraw/bug_amp. The framework supports multiple amplification strategies, including Brute-Force (*BF*), Simulated Annealing (*SA*), Genetic Algorithms (*GA*), and an ensemble classifier-based method (*Ens*). It is designed to be easily extensible, encouraging researchers and practitioners to contribute and experiment with their own search techniques. By following the provided templates and interface guidelines, users can seamlessly integrate new amplification strategies into the framework. We warmly invite the community to build upon our work and adapt the system to their specific needs. All experiments were conducted on the high-performance computing (HPC) infrastructure at Ben-Gurion University of the Negev (BGU), which is managed using an internal `SLURM` workload manager. `SLURM` allowed us to schedule thousands of concurrent and independent test executions efficiently across a cluster of multi-core servers. The use of this environment significantly accelerated the experimentation process and allowed us to evaluate all methods consistently across all problem instances. Although the experiments were originally executed in a `SLURM`-based cluster environment, the entire codebase is portable. It can be executed on any standard Linux or Windows machine or cloud platform using Python 3 and common scientific libraries, without the need for `SLURM`. The repository includes detailed instructions for reproducing the experiments, ensuring transparency and repeatability across different platforms.

Acknowledgments: During the preparation of this study, we used `ChatGPT` and `Gemini` for the purposes of searching for related work, coding, and writing. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

BF	Brute-Force
BGU	Ben-Gurion University of the Negev
CI	Continuous Integration

CLT	Central Limit Theorem
Ens	Ensemble
GA	Genetic Algorithm
GP	Genetic Programming
LLN	Law of Large Numbers
HPC	High-Performance Computing
MLP	Multi-Layer Perceptron
PCT	Probabilistic Concurrency Testing
SA	Simulated Annealing
SD	Standard Deviation
SLURM	Simple Linux Utility for Resource Management
SMOTE	Minority Over-sampling Technique
SUT	System Under Test

References

1. Gray, J. Why Do Computers Stop and What Can Be Done About It? Technical Report 85.7, Tandem Computers, Palo Alto, CA, 1985. Accessed on 16 July 2025.
2. Bakhshi, R.; Kunche, S.; Pecht, M. Intermittent Failures in Hardware and Software. *Journal of Electronic Packaging* **2014**, *136*, 011014. https://doi.org/10.1115/1.4026639.
3. Heidelberger, P. Fast simulation of rare events in queueing and reliability models. *ACM Trans. Model. Comput. Simul.* **1995**, *5*, 43–85. https://doi.org/10.1145/203091.203094.
4. Younes, H.L.; Simmons, R.G. Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation* **2006**, *204*, 1368–1409. https://doi.org/https://doi.org/10.1016/j.ic.2006.05.002.
5. Kumar, R.; Lee, J.; Padhye, R. Fray: An Efficient General-Purpose Concurrency Testing Platform for JVM. *arXiv* **2025**, *abs/2501.12618*. https://doi.org/10.48550/arXiv.2501.12618.
6. Burckhardt, S.; Kothari, P.; Musuvathi, M.; Nagarakatte, S. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10), Pittsburgh, PA, USA, 2010; pp. 167–178. https://doi.org/10.1145/1735970.1736040.
7. Zhao, H.; Wolff, D.; Mathur, U.; Roychoudhury, A. Selectively Uniform Concurrency Testing. *Proceedings of the ACM on Programming Languages (ASPLOS)* **2025**, *5*. https://doi.org/10.1145/3669940.3707214.
8. Ramesh, A.; Huang, T.; Riar, J.; Titzer, B.L.; Rowe, A. Unveiling Heisenbugs with Diversified Execution. *ACM on Programming Languages* **2025**, *9*, 393–420. https://doi.org/10.1145/3720428.
9. Godefroid, P.; Levin, M.Y.; Molnar, D.A. Effective Testing for Concurrency Bugs. Tech. rep. mpi–sws–2015–004, MPI–SWS, 2015. Accessed on 16 July 2025.
10. Han, T.; Gong, X.; Liu, J. CARDSHARK: Understanding and Stabilizing Linux Kernel Concurrency Bugs Against the Odds. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 2024; pp. 1867–1884. Accessed on 16 July 2025.
11. Bianchi, F.A.; Pezzè, M.; Terragni, V. A Search-Based Approach to Reproduce Crashes in Concurrent Programs. In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), Paderborn, Germany, 2017; pp. 221–232. https://doi.org/10.1145/3106237.3106292.
12. Rasheed, S.; Dietrich, J.; Tahir, A. On the Effect of Instrumentation on Test Flakiness. In Proceedings of the 2023 IEEE/ACM International Conference on Automation of Software Test (AST), San Francisco, CA, USA, 2023; pp. 329–341. https://doi.org/10.1109/AST58925.2023.00016.
13. Xu, J.; Wolff, D.; Han, X.; Li, J.; Roychoudhury, A. Concurrency Testing in the Linux Kernel via eBPF. *arXiv* **2025**, *abs/2504.21394*. https://doi.org/10.48550/arXiv.2504.21394.
14. Musuvathi, M.; Qadeer, S.; Ball, T.; Basler, G.; Nainar, P.A.; Neamtiu, I. Finding and Reproducing Heisenbugs in Concurrent Programs. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), San Diego, CA, USA, 2008; pp. 267–280. Accessed on 16 July 2025.

15. Shashank, S.S.; Sachdeva, J.; Mukherjee, S.; Deligiannis, P. Nekara: A Generalized Concurrency Testing Library. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 2021; pp. 634–646. https://doi.org/10.1109/ASE51524.2021.9678838.
16. Lee, S.; Zhang, H.; Viswanathan, M. Probabilistic Concurrency Testing for Weak Memory Programs. In Proceedings of the 28th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Montreal, QC, Canada, 2023; pp. 133–147. https://doi.org/10.1145/3575693.3575729.
17. Elmas, T.; Burnim, J.; Necula, G.C.; Sen, K. CONCURRIT: A Domain Specific Language for Reproducing Concurrency Bugs. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13), Seattle, WA, USA, 2013; pp. 441–452. https://doi.org/10.1145/2491956.2462162.
18. Chen, Y.; Liu, S.; Gan, Q. Effective Concurrency Testing for Go via Directional Primitive Scheduling. In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), Luxembourg, 2023; pp. 138–149. https://doi.org/10.1109/ASE56229.2023.00086.
19. Li, X.; Li, W.; Zhang, Y.; Zhang, L. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19). ACM, 2019, pp. 169–180. https://doi.org/10.1145/3293882.3330574.
20. Böttinger, K.; Godefroid, P.; Singh, R. Learn&Fuzz: Machine Learning for Input Fuzzing. *arXiv* **2018**, *abs/1701.07232*. https://doi.org/10.48550/arXiv.1701.07232.
21. Amalfitano, D.; Faralli, S.; Hauck, J.C.R.; Matalonga, S.; Distante, D. Artificial Intelligence Applied to Software Testing: A Tertiary Study. *ACM Computing Surveys* **2023**, *56*, 1–29. https://doi.org/10.1145/3616372.
22. Leesatapornwongsa, T.; Lukman, J.F.; Lu, S.; Gunawi, H.S. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In Proceedings of the 51st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16), April 2016, Vol. 51, *SIGPLAN Notices*, pp. 517–530. https://doi.org/10.1145/2954679.2872374.
23. Sipper, M.; Green, B.; Ronen, Y.; Gat, T.; Hoffman, S.; Zohar, N. EC-KitY: Evolutionary computation tool kit in Python with seamless machine learning integration. *SoftwareX* **2023**, *23*, 101381. https://doi.org/10.1016/j.softx.2023.101381.
24. Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*; Addison-Wesley: Reading, MA, USA, 1989. Accessed on 16 July 2025.
25. Karafotias, G.; Hoogendoorn, M.; Eiben, A.E. Parameter Control in Evolutionary Algorithms: Trends and Challenges. *IEEE Transactions on Evolutionary Computation* **2015**, *19*, 167–187. https://doi.org/10.1109/TEVC.2014.2308294.
26. Elyasaf, A.; Farchi, E.; Margalit, O.; Weiss, G.; Weiss, Y. Generalized Coverage Criteria for Combinatorial Sequence Testing. *IEEE Transactions on Software Engineering* **2023**, *49*, 4023–4034. https://doi.org/10.1109/TSE.2023.3279570.
27. Wasserstein, R.L.; Lazar, N.A. The ASA's Statement on p-Values: Context, Process, and Purpose. *The American Statistician* **2016**, *70*, 129–133. https://doi.org/10.1080/00031305.2016.1154108.
28. Liu, K.; Chen, Z.; Liu, Y.; Zhang, J.M.; Harman, M.; Han, Y.; Ma, Y.; Dong, Y.; Li, G.; Huang, G. LLM-Powered Test Case Generation for Detecting Bugs in Plausible Programs. *arXiv preprint arXiv:2404.10304* **2024**. https://doi.org/10.48550/arXiv.2404.10304.
29. Ouédraogo, W.C.; Plein, L.; Kaboré, K.; Habib, A.; Klein, J.; Lo, D.; Bissyandé, T.F. Enriching Automatic Test Case Generation by Extracting Relevant Test Inputs from Bug Reports. *Empirical Software Engineering* **2025**, *30*, 1–27. https://doi.org/10.1007/s10664-025-10635-z.
30. Benavoli, A.; Corani, G.; Mangili, F. Should we really use post-hoc tests based on mean-ranks? *CoRR* **2015**, *abs/1505.02288*. https://doi.org/10.48550/arXiv.1505.02288.
31. Might, M.; Horn, D.V. A Family of Abstract Interpretations for Static Analysis of Concurrent Higher-Order Programs. In *Static Analysis (SAS 2011)*; Yahav, E., Ed.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2011; Vol. 6887, *Lecture Notes in Computer Science*, pp. 180–197. https://doi.org/10.1007/978-3-642-23702-7_16.
32. Bora, U.; Vaishay, S.; Joshi, S.; Upadrasta, R. OpenMP Aware MHP Analysis for Improved Static Data-Race Detection. In Proceedings of the 7th IEEE/ACM Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC '21). IEEE/ACM, 2021, pp. 1–11. https://doi.org/10.1109/LLVMHPC54804.2021.00006.
33. Matsakis, N.D.; II, F.S.K. The Rust Language. *Ada Letters* **2014**, *34*, 103–104. https://doi.org/10.1145/2663171.2663188.

34. Godefroid, P.; Klarlund, N.; Sen, K. DART: Directed Automated Random Testing. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Association for Computing Machinery, 2005, pp. 213–223. https://doi.org/10.1145/1065010.1065036.
35. Tehrani, A.; Khaleel, M.; Akbari, R.; Jannesari, A. DeepRace: Finding Data Race Bugs via Deep Learning. *arXiv preprint arXiv:1907.07110* **2019**. https://doi.org/10.48550/arXiv.1907.07110.
36. Chen, H.; Guo, S.; Xue, Y.; Sui, Y.; Zhang, C.; Li, Y.; Wang, H.; Liu, Y. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In Proceedings of the 29th USENIX Security Symposium (USENIX Security '20), Boston, MA, USA, 2020; pp. 2325–2342. Accessed on 16 July 2025.
37. Roemer, J.; Genç, K.; Bond, M.D. SmartTrack: Efficient Predictive Race Detection. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20). ACM, 2020, pp. 747–762. https://doi.org/10.1145/3385412.3385993.
38. O'Callahan, R.; Jones, C.; Froyd, N.; Huey, K.; Noll, A.; Partush, N. Engineering Record And Replay For Deployability. In Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17). USENIX Association, 2017, pp. 377–390. Extended technical report available via arXiv; accessed 16 July 2025.
39. Holzmann, G.J. The Model Checker SPIN. *IEEE Transactions on Software Engineering* **1997**, *23*, 279–295. https://doi.org/10.1109/32.588521.
40. Clarke, E.M.; Biere, A.; Raimi, R.; Zhu, Y. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design* **2001**, *19*, 7–34. https://doi.org/10.1023/A:1011276507260.
41. Clarke, E.M.; Grumberg, O.; Jha, S.; Lu, Y.; Veith, H. Counterexample-Guided Abstraction Refinement. In Proceedings of the 12th International Conference on Computer Aided Verification (CAV). Springer, 2000, Vol. 1855, *Lecture Notes in Computer Science*, pp. 154–169. https://doi.org/10.1007/10722167_15.
42. Namjoshi, K.S.; Trefler, R.J. Parameterized Compositional Model Checking. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016), 2016, Vol. 9636, *Lecture Notes in Computer Science*, pp. 589–606. https://doi.org/10.1007/978-3-662-49674-9_39.
43. Legay, A.; Lukina, A.; Traonouez, L.; Yang, J.; Smolka, S.A.; Grosu, R. Statistical Model Checking. In *Computing and Software Science*; Springer Cham, 2019; Vol. 11506, *Lecture Notes in Computer Science*, pp. 478–504. https://doi.org/10.1007/978-3-319-91908-9_23.
44. Xu, M.; Kashyap, S.; Zhao, H.; Kim, T. KRACE: Data Race Fuzzing for Kernel File Systems. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020, pp. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078.
45. Lu, S.; Park, S.; Seo, E.; Zhou, Y. 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 (ASPLOS '08), Seattle, WA, USA, 2008; pp. 329–339. https://doi.org/10.1145/1346281.1346323.
46. Musuvathi, M.; Qadeer, S.; Ball, T.; Basler, G.; Engler, D.R.; Foster, J.C.; Ghosh, A.K. Finding and Reproducing Heisenbugs in Concurrent Programs. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA, 2008; pp. 267–280. Accessed on 16 July 2025.
47. Tian, Y.; Yu, Y.; Wang, P.; Zhou, R.; Jin, H.; Xie, T. RACEBENCH: A Benchmark Suite for Data Race Detection Tools. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11). ACM, 2011, pp. 142–151. https://doi.org/10.1145/2025113.2025136.
48. Zhang, W.; Yao, C.; Lu, S.; Huang, J.; Tan, T.; Liu, X. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11). ACM, 2011, pp. 251–264. https://doi.org/10.1145/1950365.1950395.
49. Lin, Z.; Marinov, D.; Zhong, H.; Chen, Y.; Zhao, J. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs. In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15). IEEE / ACM, 2015, pp. 178–189. https://doi.org/10.1109/ASE.2015.87.
50. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14), San Jose, CA, USA, July 2014; pp. 437–440. https://doi.org/10.1145/2610384.2628055.
51. Madeiral, F.; Urli, S.; de Almeida Maia, M.; Monperrus, M. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19). IEEE, 2019, pp. 468–478. https://doi.org/10.1109/SANER.2019.8667991.

doi:10.20944/preprints202507.2304.v1

52. Karampatsis, R.; Sutton, C. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR ’20). ACM, 2020, pp. 573–577. https://doi.org/10.1145/3379597.3387491.
53. Tu, T.; Liu, X.; Song, L.; Zhang, Y. Understanding Real-World Concurrency Bugs in Go. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’19). ACM, 2019, pp. 865–878. https://doi.org/10.1145/3297858.3304069.
54. Yuan, T.; Li, G.; Lu, J.; Liu, C.; Li, L.; Xue, J. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In Proceedings of the 18th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO ’21). IEEE / ACM, 2021, pp. 187–199. https://doi.org/10.1109/CGO51591.2021.9370317.
55. Torres, C.F.; Iannillo, A.K.; Gervais, A.; State, R. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In Proceedings of the 2021 IEEE European Symposium on Security and Privacy (EuroS&P ’21). IEEE, 2021, pp. 213–228. https://doi.org/10.1109/EuroSP51992.2021.00018.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.