Unveiling Heisenbugs with Diversified Execution

ARJUN RAMESH, Carnegie Mellon University, USA
TIANSHU HUANG, Carnegie Mellon University, USA
JASPREET RIAR, Carnegie Mellon University, USA
BEN L. TITZER, Carnegie Mellon University, USA
ANTHONY ROWE, Carnegie Mellon University, Bosch Research, USA

Heisenbugs, notorious for their ability to change behavior and elude reproducibility under observation, are among the toughest challenges in debugging programs. They often evade static detection tools, making them especially prevalent in cyber-physical edge systems characterized by complex dynamics and unpredictable interactions with physical environments. Although dynamic detection tools work much better, most still struggle to meet low enough jitter and overhead performance requirements, impeding their adoption. More importantly however, dynamic tools currently lack metrics to determine an observed bug's "difficulty" or "heisen-ness" undermining their ability to make any claims regarding their effectiveness against heisenbugs.

This paper proposes a methodology for detecting and identifying heisenbugs with low overheads at scale, actualized through the lens of dynamic data-race detection. In particular, we establish the critical impact of *execution diversity* across both instrumentation density and hardware platforms for detecting heisenbugs; the benefits of which outweigh any reduction in efficiency from limited instrumentation or weaker devices. We develop an experimental WebAssembly-backed dynamic data-race detection framework, Beanstalk, which exploits this diversity to show superior bug detection capability compared to *any* homogeneous instrumentation strategy on a fixed compute budget. Beanstalk's approach also gains power with *scale*, making it suitable for low-overhead deployments across numerous compute nodes. Finally, based on a rigorous statistical treatment of bugs observed by Beanstalk, we propose a novel metric, the *heisen factor*, that similar detectors can utilize to categorize heisenbugs and measure effectiveness. We reflect on our analysis of Beanstalk to provide insight on effective debugging strategies for both in-house and in deployment settings.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Empirical software validation*; • **Computer systems organization → Embedded software**.

Additional Key Words and Phrases: heisenbugs, data-races, WebAssembly, cyber-physical systems, statistical bug analysis, heisen factor

ACM Reference Format:
Arjun Ramesh, Tianshu Huang, Jaspreet Riar, Ben L. Titzer, and Anthony Rowe. 2025. Unveiling Heisenbugs with Diversified Execution. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 94 (April 2025), 28 pages. https://doi.org/10.1145/3720428

1 Introduction

*"Not only is the Universe stranger than we think, it is stranger than we **can** think"*
— Werner Heisenberg, *Across the Frontiers*

Authors' Contact Information: Arjun Ramesh, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, arjunr2@andrew.cmu.edu; Tianshu Huang, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, tianshu2@andrew.cmu.edu; Jaspreet Riar, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, riarjaspreet77@gmail.com; Ben L. Titzer, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, btitzer@andrew.cmu.edu; Anthony Rowe, Carnegie Mellon University, Bosch Research, Pittsburgh, Pennsylvania, USA, agr@andrew.cmu.edu.

ACM 2475-1421/2025/4-ART94
https://doi.org/10.1145/3720428

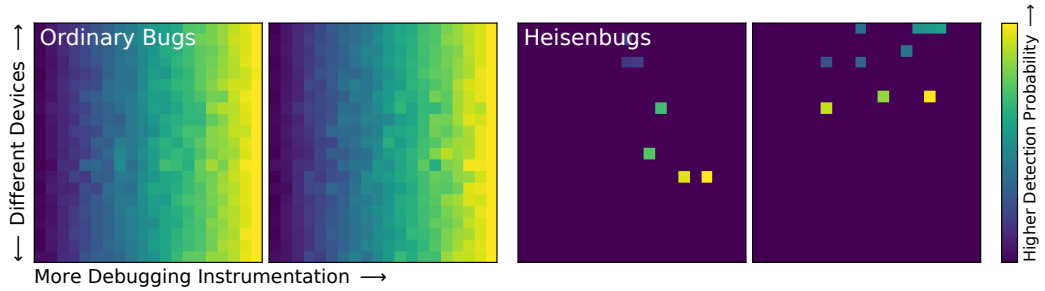

Fig. 1. The probability of detecting two pairs of example bugs in our dataset across different devices and amount of debugging instrumentation. *Ordinary bugs* behave similarly across devices, and become easier to detect with more debugging instrumentation, while *Heisenbugs* can only be detected with specific device-instrumentation configurations and disappear if too much or too little debugging instrumentation is added.

Heisenbugs are a programmer's worst nightmare. Aptly titled as such, heisenbugs "magically" alter their behavior in response to subtle differences in a program's execution context (Fig. 1). These elusive bugs often change or even disappear in the presence of minimally-intrusive debuggers, almost as if they possess properties that stymie attempts to reproduce or debug them. Their insidious nature can also cause rare system failures far in the future that make root cause analysis especially difficult [20]. Resolving heisenbugs thus demands intense manual debugging efforts and an exploration of a myriad of possible execution interleavings caused by subtle timing differences.

Heisenbugs are especially problematic in modern cyber-physical edge systems [11, 31, 48, 74], which often involve distributed tasks across heterogeneous hardware and perform sensing/actuation in unpredictable physical environments during deployment. Static analysis tools [17, 21, 36, 52, 76] fail to catch these bugs, and are challenging to incorporate into many developer workflows [39] since they require source code and enumeration of synchronization primitives used by the application. Furthermore, increased computational capabilities have pushed more complexity into safety-critical domains where the repercussions of software-related system failures can be severe, such as medical [78] and automotive systems [3, 62]. As the embedded landscape shifts from primarily single-core processors to more complex multicore systems [23], developers are also encountering a new wave of concurrency bugs – even with previously tested legacy software – that were formerly unobservable due to implicit synchronization enforced by the original OS scheduler. Large-scale deployment of such edge systems, coupled with a push for increased software autonomy thus makes effective testing and debugging critical for future applications in these ecosystems.

This paper examines heisenbug detection through the lens of *dynamic data-race detection* using delay-injection techniques [24, 63], which have shown promise in controlled environments (e.g. Microsoft [49]) for their generalizability to various programming paradigms and ease of integration. We find that such debuggers, while seemingly effective, cannot be objectively evaluated, primarily because they lack a robust *evaluation metric*. Most existing solutions focus solely on the number of bugs detected, neglecting to account for the *difficulty* of bugs found or missed. This omission undermines the ability to make convincing comparative assessments of any debugger's effectiveness – especially against the most critical heisenbugs – and applies to many bug classes beyond data-races.

To concretize this observation, consider the `fibonacci` benchmark example (Fig. 2) which contains multiple data-races as the two unique threads execute lines `9-14`. An initial inspection might suggest that all these simple data-races in the same code block behave uniformly, as observed for the top row of bugs `[(13,13),(12,12),(10,10)]`. In reality, however, the bottom row of bugs

```c
#define N 20000
// Global State
long fibonacci[2] = {1, 0};
int num_ct = 1;
volatile int p = 0;

void *fib_thread(void *tid) {
  int tnum = *(int *)tid;
  while (num_ct < N) {
    p += tnum;
    long f1 = fibonacci[1];
    fibonacci[1] = fibonacci[0];
    fibonacci[0] += f1;
    num_ct++;
  }
  return NULL;
}

int main() {
  // Spawn + join 2 threads
  // invoking `fib_thread(tid)`
  spawn_threads(2, fib_thread);
  return 0;
}
```

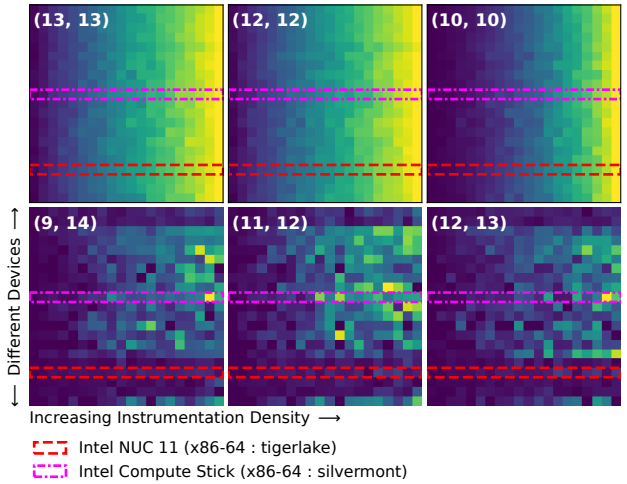

Fig. 2. Detectability profiles for a subset of data-races in the `fibonacci` benchmark. The pair of racing lines of code are shown in the top-left of each profile. Common intuition would suggest that all these 'ordinary' bugs in the same code block (`while` loop) would manifest similarly. Surprisingly however, some of these bugs exhibit heisenbug properties (bottom row) and are highly susceptible to subtle device + instrumentation characteristics – the *Compute Stick* offer far superior detectability than the *NUC* for these bugs.

`[(9,14),(11,12),(12,13)]` demonstrate significant "heisen-ness" – they are virtually undetectable on the *NUC 11* at any level of instrumentation while simultaneously well detectable at arbitrary instrumentation points on the *Compute Stick*. Note our distinction between a bug's overall rarity and it's "heisen-ness" – common bugs can still be heisenbugs if they are highly susceptible to the execution environment (the above example), and likewise rare bugs can behave uniformly across environments. Developing an accurate analytical interpretation for such quirky heisenbug behavior is infeasible for several reasons:

- *Compiler Optimizations*: Layers of intricate optimization passes (LICM, PRE, etc.) can hide or expose seemingly obvious bugs from being detected.
- *System Stack Complexity*: Analytical models of hardware and OS effects, commonly explored for worst-case execution time [6], are usually overly simplistic or inaccurate due to numerous intertwined layers of non-determinism.

Edge environments introduce additional unique challenges, including stringent timing and jitter overhead constraints, along with embeddability limitations, which frequently impair most debuggers' efficiency and practicality in a standard development/deployment workflow. We thus identify three core requirements for an effective debugger – vital across all domains but especially crucial at the edge:

Heisenbug Detection Effectiveness: Finding "difficult" bugs that evade simple in-house testing is arguably the most valuable quality of a debugger. Notably, these tend to *predominantly comprise of heisenbugs* (shown in Fig. 9) since "ordinary" bugs are low-hanging fruit usually resolved by simple testing and static detectors. An ideal debugger will identify bugs originating from convoluted chains of events that are otherwise unobservable by analysis-based techniques yet likely to appear during deployment. It hence must explore a wide range of program state and execution interleavings within

a short fixed compute budget. Evaluating this property also requires a formal characterization of the "heisen-ness" of a bug.

Tunability and Non-Intrusive Deployability: Many applications' functionality, especially soft real-time or safety-critical ones, depend on maintaining specific timing or jitter guarantees during execution. An ideal debugger would be able to operate even under such conditions, providing *predictable* yet *tunable* overheads to allow a range of applications to obtain the best effectiveness while still operating within any slack time available for the application.

Flexibility And Scalability: Integrating a debugger into existing workflows presents a subtle yet significant challenge that is crucial for its adoption. Given the absence of standardized software infrastructure across different domains and cyber-physical edge systems, an ideal debugger must be language-agnostic and capable of scaling effortlessly across diverse hardware platforms. Most importantly, it is crucial for these techniques to be *sound and fully automated*, eliminating the need for programmer intervention to annotate instrumentation points or verify false positives, both of which are laborious and expensive in practice.

Towards these objectives, we propose a methodology for detecting and identifying heisenbugs with low overheads at scale. This methodology is realized through *Beanstalk*[1], an experimental data-race detection framework that is highly effective against heisenbugs, offers easily tunable overheads suitable during deployment and can support applications across numerous domains with no false positives. We provide a statistical evaluation of Beanstalk and propose the *heisen factor* – a metric quantifying the susceptibility of a bug's observability to changes in the underlying execution environment (i.e, its "heisen-ness") – that is generalizable across similar classes of debuggers. To realize these outcomes, the methodology behind Beanstalk incorporates the following solutions:

Partial Instrumentation: Beanstalk is end-to-end automated and does not require any manual annotations to reduce instrumentation overhead. Instead, we partially instrument a randomly selected subset of memory accesses in the program, specified as a percentage called the *instrumentation density*. Instrumentation density is a single parameter, allowing applications to easily tune the maximum instrumentation density to the max timing slack they can operate under (see Fig. 16). While low density usually comes at the cost of decreased effectiveness, we compensate by *scaling-out* partially instrumented programs across many devices. With enough scale-out, lower density instrumentation can then provide coverage comparable or even superior to high density instrumentation on a single device, enabling active detection during deployment even on weak embedded devices.

Execution Diversity: High instrumentation load makes each data-race more likely to be instrumented, which should lead to higher detectability across the board. While this holds for many common bugs, many heisenbugs — especially rare ones — are similarly detectable or even more detectable at specific instrumentation loads *lower than the maximum* (Fig. 1). This indicates that implicit timing effects introduced by instrumentation can greatly impact a bug's observability, meaning that *more instrumentation is not always the optimal debugging strategy*. As a corollary, platform heterogeneity – executing programs across devices with diverse hardware configurations (CPUs, OS, etc.) – can induce another dimension of side effects, playing a significant role in the observability of some bugs. We take advantage of this diversity insight in Beanstalk to observe heisenbugs more effectively within a fixed compute budget by varying the instrumentation load and running across different platforms within any slack-time the application can tolerate.

[1]The name Beanstalk is inspired by Jack & the Beanstalk as a tool that bridges the cloud (powerful devices) to ground (weaker edge devices)

WebAssembly (Wasm): Wasm [35] is a cross-platform, low-level bytecode that serves as a polyglot compilation target with high-performance in-memory sandboxing. Given these properties, Wasm offers an excellent standardized executable format for safety, distribution, and runtime code analysis, allowing programs to be easily instrumented and deployed flexibly across heterogeneous platforms. Unlike common bytecode options like Java or .NET, Wasm's low memory overhead, determinism, and avoidable garbage collection make it a suitable bytecode format for timing-constrained embedded and edge devices. Beanstalk instruments Wasm programs at the granularity of load/store memory accesses to detect data-races, which can be traced back to the source code[2].

Contributions: We design a diversity-driven methodology for efficient heisenbug detection realized through Beanstalk, an end-to-end data-race debugging framework that uses varying instrumentation across heterogeneous devices. Beanstalk requires no programmer intervention and leverages Wasm for seamless scalability. We summarize our contributions as follows:

(1) **Diversity-Guided Scalable Instrumentation**: We leverage *partial* program instrumentation to enable debugging under restricted compute and timing constraints, and scale across many heterogeneous devices to compensate for detection coverage (Sec. 4).
(2) **Heisenbug Analysis**: We define the *heisen factor* (Sec. 5.2), which quantifies the susceptibility of a bug's behavior to changes in instrumentation and the execution platform, and characterize the behavior of 109 bugs observed by Beanstalk (Sec. 5.3). To the best of our knowledge, we are the first to quantitatively study these effects in a statistical manner.
(3) **Evaluation**: We use Beanstalk to evaluate the value of execution diversity, which finds significantly more heisenbugs than a typical homogeneous baseline across all compute budgets, often surpassing the latter even when handicapped by timing constraints (Sec. 5.4).

While the techniques in our framework can be extended to targets beyond Wasm, the scalability, instrumentation ease, and language flexibility provided by Wasm make it an ideal computational model for cross-platform deployment and debugging.

2 Background

This section outlines key background knowledge relevant to the techniques utilized in Beanstalk, focusing on dynamic data-race detection and Wasm in context of the edge systems.

2.1 Data-Race Detection

Data-races, in the context of concurrent programs, occur when a shared memory resource is *simultaneously* and *non-atomically* accessed by multiple *threads*, with at least one performing a write access. Many modern language abstractions (e.g., safe Rust [44], Dala [26]) and software runtimes (e.g., Grace [12]) provide formal guarantees of data-race freedom at compile-time and runtime respectively. However, the constraints imposed by these programming paradigms coupled with high cost of modernizing legacy codebases have limited their proliferation, particularly in edge-embedded systems [57] and automotive environments [43]. This underscores the need for specialized data-race detection solutions tailored to these domains.

Data-race detection tools can analyze code either *statically* (prior to execution) or dynamically (at runtime). Static detectors usually use context- or flow-sensitive analysis on the control-flow graph of source code [40, 65] or bytecode [13, 60]. However, since this is generally an undecidable problem [46, 47], they often make aggressive or conservative assumptions on validity of detected races, which increases effort in sifting through false positives or provides minimal true positives

[2]Support for DWARF symbols in Wasm toolchains are under development and will be integrated into this work once stable.

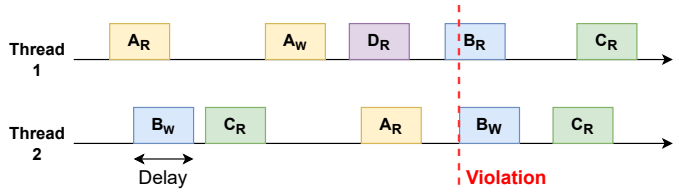

Fig. 3. Illustration of the delay-injection algorithm methodology for reads (**R**) and writes (**W**) on a set of resources (**A** – **D**). $\mathbf{B_R}$ and $\mathbf{B_W}$ have simultaneously overlapping delay windows, producing a data-race.

respectively. Dynamic detectors [34] often provide higher quality detection by using leveraging runtime information, and utilize various techniques such as:

Happens-before Analysis: Happens-before (HB) analysis, employed by algorithms such as Djit$^+$ [64] and FastTrack [27], is a common technique that uses vector-clocks [45] to infer concurrent accesses to shared variables that could be triggered simultaneously by different threads. HB analysis is typically complete (no false positives), but often incurs high overheads for tracking concurrent accesses to all variables and requires knowledge of all synchronization primitives to properly increment clocks and avoid false negatives.

Lockset Analysis: Lockset analysis [19, 70, 77] traces lock acquisitions/releases, flagging accesses to shared variables from different threads as potential data-races if they do not share an acquired lock. Lockset analysis is less computationally expensive than HB analysis, but requires knowledge of the underlying locking discipline, and reports false positives for alternative synchronization techniques (e.g. fork-join or barrier synchronization). Some popular tools such as ThreadSanitizer [73] and Racetrack [84] use a hybrid of lockset and HB analysis for improved precision and efficiency.

Delay-Injection Detection: Delay-injection, adopted by Beanstalk, differs from prior techniques as an *observation-oriented* (as opposed to analysis-oriented) approach which minimizes runtime analysis and instead intends to immediately catch data races as they occur. This mechanism injects a delay (e.g. using a bounded spin-loop) for a small window before an access (Fig. 3), during which they probe for concurrent thread-unsafe accesses to flag as data-races. This method is complete (no false positives) and avoids a reliance on detailed interleaving analysis or knowledge of synchronization primitives, making it practically viable. However, delay-injection relies on data-races both occurring and being probed, making a balanced optimization between *quality* and *overhead* an important design goal to ensure effective results with minimal probing overhead.

DataCollider [24] employs random sparse sampling of static instructions and hardware-specific breakpoint registers to minimize overhead. While effective, this approach does not scale well to edge systems with heterogeneous hardware. TSVD [49], in contrast, utilizes programmer annotations, instrumentation of portable .NET class methods, and happens-before inference to strategically reduce delay overhead. While highly effective within a controlled environment like Microsoft, .NET's dependence on garbage collection and manual object-level API annotations in TSVD for thread safety is costly, error-prone, and impractical when applied to a broader edge ecosystem with third-party vendors. Moreover, neither approach explores the potential benefits of incorporating execution diversity incorporated by Beanstalk, which could significantly enhance their effectiveness against elusive heisenbugs that have yet to manifest.

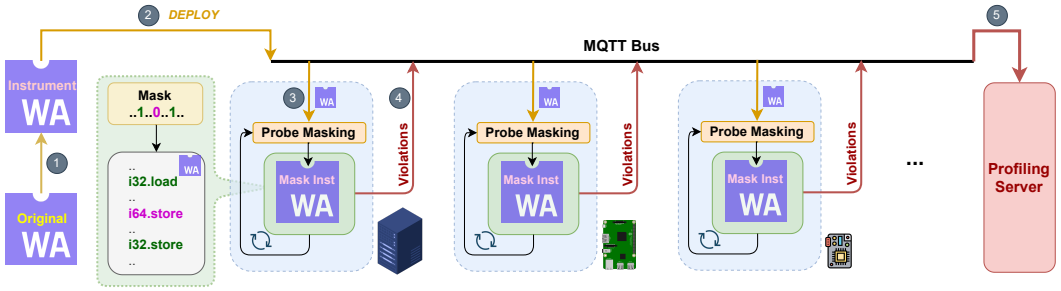

Fig. 4. The Beanstalk system architecture scales across heterogeneous platforms. Debugging begins by (1) instrumenting memory accesses in Wasm binaries, which are (2) deployed to a heterogeneous cluster. (3) Beanstalk Engines on each cluster node then execute each program with a *probe mask*, and (4) report any thread-safety violations discovered, which are (5) logged to a profiling server.

2.2 WebAssembly

WebAssembly(Wasm) is a portable bytecode originally developed as an efficient in-memory sandboxed execution format for the Web [35]. With near-native execution speeds, low memory overheads, and polyglot compilation targetability, Wasm has gained popularity for lightweight virtualization in both cloud and edge systems [29, 58, 61], demonstrating feasibility even for resource-constrained embedded devices [38, 71, 79]. Wasm also includes structured control flow, type-checking, and a stack-based execution model that cannot be addressed in memory, protecting applications against RCE attacks or other security exploits by design. Wasm's properties, coupled with efforts towards various system interfaces [9, 67] and domain-specific profiles [1], thus enable it to penetrate many edge ecosystems such as industrial automation and automotive software [68].

As an easily scalable, narrow-waist abstraction over extensive ISAs and languages, Wasm shows great potential to operate as a unified code analysis platform, simplifying software workflows. At the time of writing, popular Wasm engines report around an average of 2x slowdown [22] over native code. However, compiler and engine optimizations for Wasm are an extensive research topic [4, 80] that has made great strides with new language extensions and sandboxing optimizations [83].

3 Beanstalk

Beanstalk consists of a fully automated end-to-end pipeline to deploy programs, detect violations, and analyze consolidated results across a host of devices (Fig. 4). Beanstalk leverages Wasm's platform independence to operate transparently across heterogeneous nodes with different resource capacities, making it ideal for edge deployments. Wasm binaries are first instrumented for data-race detection using a custom *bytecode rewriting* tool prior to deployment, allowing nodes flexibility in the selection of Wasm engines and execution tiers (interpreted, just-in-time, or ahead-of-time compiled). A centralized orchestrator then deploys the instrumented binaries to nodes connected over a lightweight MQTT pub/sub network. Each node contains a Beanstalk Engine which is locally capable of probing a select subset of memory accesses by generating a *probe mask*; this allows for effective local instrumentation schemes (e.g. random subsets of probes) without requiring any modification to the binary itself. Finally, after running for a desired compute budget, unique *violations* are shepherded to a profiling server for consolidated analysis.

Beanstalk allows flexibility in choosing a data-race detection algorithm that operates on static memory-access instructions. We implement a simple trapped-delay injection algorithm (Sec. 4.4); while similar to other detection algorithms like TSVD, we do not modify instrumentation during

execution or insert probabilistic delays, which can cause unreliable timing jitter to applications, and instead enable low-overhead with partial scalable instrumentation.

4 Beanstalk Implementation

We break down our implementation of Beanstalk, which instruments Wasm memory accesses to find data races (Sec. 4.1), into three phases: instrumentation using *bytecode rewriting* (Sec. 4.2), deployment to the Beanstalk Engine using *probe masking* (Sec. 4.3), and finally execution of the online data-race detection algorithm (Sec. 4.4). We also provide optimizations that may be performed prior to this to "prune" the set of observed accesses (Sec. 4.5).

4.1 Memory Access Instrumentation

Edge applications generally lack object-oriented programming models, which makes method annotation-driven probing (e.g. TSVD) difficult. We instead instrument all low-level *memory accesses*, increases coverage to all possible memory-based data-race conditions. While this comes at the cost of requiring more probes, we address this using (stochastic) partial instrumentation (Sec. 4.3). Unlike memory address instrumentation [24], memory access instrumentation size is more memory efficient since it is statically determined and independent of memory usage or the dynamic memory accesses, both of which may grow exponentially large. The use of Wasm in Beanstalk also provides several advantages for instrumenting memory accesses:

(1) Wasm's execution model uses a private, virtual execution stack per thread meaning stack accesses need not be probed. We only probe instructions that operate on shared memory.
(2) Memory accesses are explicit, eliminating the need for accountability of complex memory side-effects from non-load/store instructions.
(3) Wasm offers only sequential consistency for atomic operations with no weaker variants.

All memory operations also currently avoid indirect addressing modes, consistently using a *base+offset* addressing mode, making the Wasm bytecode an easy instrumentation target.

4.2 Instrumentation Injection using Bytecode Rewriting

Although dynamic instrumentation allows for more complex capabilities, we employ *static instrumentation* for Beanstalk in order to enable execution on *any* Wasm engine. Static instrumentation embeds logic directly into the target Wasm bytecode [56] offline prior to execution, eliminating injection costs at runtime. Statically inserted logic is, by design, more susceptible to altering program behavior and thus careful implementation is required to ensure non-intrusiveness. However, unlike dynamic instrumentation, it requires very little runtime support, allowing instrumented binaries to be both backward and forward compatible with all Wasm engines. Currently, dynamic instrumentation is supported only by the Wizard [4] engine, which would limit Beanstalk's practicality. Should this capability become standardized in the Wasm ecosystem however, we can seamlessly replace it in place of static instrumentation without altering Beanstalk's overall design.

Beanstalk instruments Wasm modules using a custom-designed *bytecode rewriting* tool, which provides an instrumentation pass that allocates memory for control metadata and injects probes to test for data-races (Fig. 5). Static instrumentation for Wasm also offers two subtle advantages:

- Wasm statically links all files into a single executable binary, allowing for complete code coverage comparable to dynamic instrumentation techniques.
- Instrumentation logic is reasonably simple and bytecode rewriting allows static validation of the module logic (e.g. type-safety) prior to deployment.

Our exceptionally lightweight bytecode rewriter is integrated into the Beanstalk Engine, facilitating *just-in-time* instrumentation of binaries locally, even on weaker devices, prior to loading

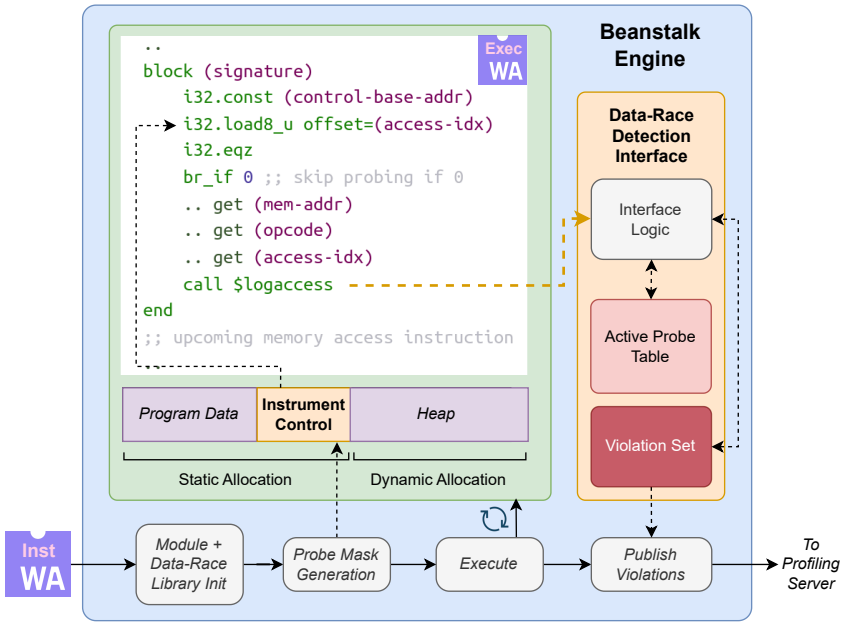

Fig. 5. Beanstalk Engine software architecture for probe activation and data-race detection.

the program. This capability allows us to entirely exclude the producer from the trusted chain, providing a important advantage when operating on untrusted Wasm binaries.

4.3 Dynamic Partial Instrumentation using Probe Masking

Beanstalk achieves low execution overhead with effective bug detection coverage by partial instrumentation of random subsets of probes. Probe site selection can be done naively by creating many versions of a binary with fixed instrumentation points. However, scaling up this solution is infeasible as it consumes excessive disk and network bandwidth, while simultaneously limiting the flexibility of instrumentation configurations. Instead, the Beanstalk Engine uses a *probe mask* in the instrumented binary to dynamically select which probes to activate (Fig. 5).

The probe mask is a statically-allocated array within the Wasm program's memory space with one-byte per static memory access instruction. While a probe mask-based approach adds a branching overhead to every memory access, we only need to deploy a single Wasm binary to all Beanstalk Engines, allowing each host the flexibility to activate different probes. In particular, local probing schemes like choosing a random subset of probes (used in our evaluation) can be performed independently by each Beanstalk Engine, enabling seamless scalability. During deployment in the wild with data-race detection enabled, these probe points can be statically sampled, eliminating branching overhead associated with probe masks.

4.4 Data-Race Detection Algorithm

The Beanstalk architecture is compatible with any data-race algorithm which instruments static memory-access instructions within an executing module. Allowing effortless updates/improvements to the core data-race algorithm without modifying deployed applications is hence of prime importance to Beanstalk. We thus focus on the following objectives:

Algorithm 1 Pseudo-Code for Data-Race Detection with Delay Injection (as visualized in Fig. 3)

```
   Global State  V, T, μ                ▷ V: detected violations set;  T: access table;  μ: delay cycles
   Global Init   V = {}
                 T[α] = {.mutex = unlocked, .access = ϕ .probe = False} ∀ addresses α
                 μ = 50                                  ▷ Default value used; can be user configured
 1: procedure LOGACCESS(tid, addr, access_idx)
 2:     entry ← &T[addr]                                ▷ Reference to observed address probe state
 3:     cur_access ← {.tid = tid, .access_idx = access_idx}
 4:     LOCK entry.mutex
 5:     if AtomicExchange(entry.probe, True) then       ▷ Activate probe; setup trap if previously inactive
 6:         entry.access ← cur_access                   ▷ Setup new probed access within the table
 7:         UNLOCK entry.mutex
 8:         DELAY(μ)                                    ▷ Delay window to catch concurrent collisions
 9:         entry.probe ← True                          ▷ Deactivate probe
10:     else                                            ▷ If old probe is active, collision
11:         tid_diff ← entry.tid != cur_access.tid
12:         race_req ← hasWriteAndNonAtomic(entry.access, cur_access)
13:         if tid_diff and race_req then
14:             V ← V ∪ (entry.access, cur_access)      ▷ Violation detected; record in global set
15:         end if
16:         UNLOCK entry.mutex
17:     end if
18: end procedure
```

Separation of Concerns After writing the probe mask, the Wasm program is prepared for data-race detection. Each probe activated by the probe mask invokes the `logaccess` function via the Wasm engine's foreign function interface, which is implemented within the Beanstalk Engine environment. While implementing the algorithm within the Wasm execution environment may reduce the calling overhead, decoupling the core data-race detection algorithm from it has the following key advantages:

(1) The core algorithm can be tailored to deployments without recompiling the application.
(2) The sandboxed program cannot tamper with any data-race metadata or logic[3].
(3) Program crashes (likely if critical bugs occur) do not take down the detection infrastructure.

When the `logaccess` method is called for active probes, the Wasm program forwards any metadata associated with the access – its thread ID, target memory address, and a unique static identifier for the instruction – to its implementation in the Beanstalk Engine, which performs the data-race detection algorithm (Alg. 1). This algorithm tracks the probe access metadata for entire Wasm memory in T, and considers the relevant probe state `entry` (line 2) and `cur_access` (line 3) on a memory access, performing one of the following:

- If the *entry* probe is unset, set the probe, delay μ cycles, and clear the probe (lines 6–9).
- If the *entry* probe is set, we have a true positive data-race – record the pair of accesses `entry.access` and `cur_access` as a violation to report.

The delay is implemented with loop around `noop` that occurs μ times, where μ is user configurable (see Sec. 5.1). Demand paging allows this method to only use memory as needed to store T.

[3]Wasm supports multiple isolated memories in a program that can be used to separate privileged instrumentation control and application data.

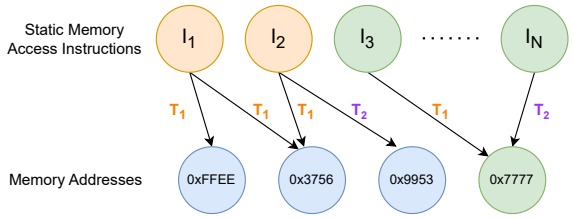

Fig. 6. We can visualize the shared access profile of an application as a bipartite graph, where each dynamic memory access from a thread represents an edge between instructions and the memory address space of the program; **shared accesses** consist of addresses with multiple accesses from different threads.

Latency Minimization Frequent calls to `logaccess` occur in most programs and may result in high overheads if implemented naively. Delay cycles are thus performed using a volatile noop loop in user-space since the base overhead for kernel calls are too expensive for each access. Furthermore, most calls to `logaccess` do not trigger violations (lines 9–16) on their same memory address, so we prioritize latency minimization of the delay injection path. Updates to `entry` however need to be performed atomically (with a mutex) to avoid concurrency bugs in the data-race algorithm itself.

4.5 Probe Set Pruning

Naively probing all accesses can introduce inefficiency at deployment, since not all accesses are fundamentally capable of causing data-races. Many data-race algorithms prune out probes to these "unshared" accesses prior to data-race detection, either statically or dynamically, such as those to thread-local variables, read-only globals, or stack variables that are not aliased by pointers across threads. We detail seamless support for this *dynamically* within our same framework, particularly using our same partial instrumentation methodology to allow tunability and low runtime overheads.

Shared Access Detection The shared access detection problem is simpler than data-race detection, since the dynamic accesses do not have to occur simultaneously to be flagged. This problem can be fundamentally condensed to a bipartite graph (Fig. 6) with edges between memory access instructions and memory addresses for a dynamic access from a specific thread. Conveniently, we can avoid re-instrumentation and use the *same* instrumented Wasm binary (Sec. 4.2) for both data-race detection and shared access detection by purely changing the `logaccess` implementation within the Beanstalk Engine. Our shared detection algorithm (Alg. 2) uses a table T, similar to Alg. 1, to serialize accesses to target memory addresses and notably tracks two pieces of state:

- $I[addr]$: The set of memory accesses instructions that have been observed to target $addr$.
- $S[addr]$: A bit / boolean flag that specifies if $addr$ is a shared address, that is, instructions from different threads have accessed $addr$ in the past.

The set P accumulates a set of true positive instructions that interact with the shared state in the program, which can serve as the probe set. P is computed in Alg. 2 based on three possible states of the memory address $addr$ targeted by the current instruction:

- *Exclusive Access* (line 5): Only one thread has accessed $addr$ so far. Accumulate current instruction into $I[addr]$ – if/once $addr$ becomes shared, $I[addr]$ gets added to P.
- *Transition Access* (line 8): First access to $addr$ from a second unique thread. Transition this $addr$ to a shared state ($S[addr] = True$), and add both the current instruction and its history $I[addr]$ to P. Once marked as shared, $addr$ remains shared until the end of execution.
- *Shared Access* (line 9): Access to a shared address. Since $addr$ is known to be shared, $I[addr]$ is unnecessary and the current instruction can be added directly to P.

Algorithm 2 Pseudo-Code for Shared Access Detection (as visualized in Fig. 6)

Global State P, I, T, S ▷ P: detected shared addresses set; I: history of instructions ▷ S: bitmap flagging shared status of address; T: access table

Global Init $P = \{\}$
$I[\alpha] = \{\}\ \forall\ addresses\ \alpha$
$T[\alpha] = \{.mutex = unlocked,\ .tid = \phi\}\ \forall\ addresses\ \alpha$
$S[\alpha] = False\ \forall\ addresses\ \alpha$

1: **procedure** LOGACCESS($tid, addr, access\_idx$)
2: $entry \leftarrow \&T[addr]$
3: LOCK $entry.mutex$
4: $tid\_diff \leftarrow entry.tid\ != tid$
5: **if** $!S[addr]$ and $!tid\_diff$ **then** ▷ *Exclusive Access*
6: $I[addr] \leftarrow I[addr] \cup \{access\_idx\}$ ▷ Record to instruction history
7: $entry.tid = tid$ ▷ Mark the exclusive thread that made the access
8: **else if** $!S[addr]$ and tid_diff **then** ▷ *Transition Access*
9: $P \leftarrow P \cup I[addr] \cup \{access\_idx\}$ ▷ Flush instruction history to P
10: $S[addr] \leftarrow True$ ▷ Future accesses to this address are shared by default
11: $I[addr] \leftarrow \{\}$
12: **else** ▷ *Shared Access*
13: $P \leftarrow P \cup \{access\_idx\}$ ▷ Bypass $I[addr]$ and add directly to P
14: **end if**
15: UNLOCK $entry.mutex$
16: **end procedure**

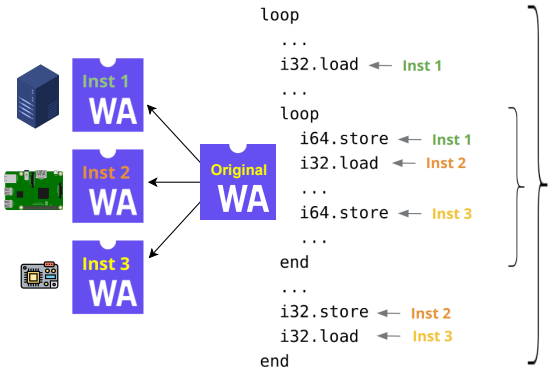

Fig. 7. Distributed Profiling for Deterministic Applications. Each **loop** scope is partitioned with an equal instrumentation weight and disjoint subsets of probes for every Wasm program instance.

For devices that are too memory constrained to maintain the full instruction history I for the max memory size of the Wasm module, approximations can be performed using a fixed size instruction history in memory with a least-recently-used (LRU) replacement strategy. Finally, just like with data-race detection, we can apply our *diversity-guided scalable instrumentation* methodology to the shared access detection with a random subset of probes to minimize both false negatives and instrumentation overhead. The final set is then computed as $P_F = \bigcup_{i=1}^{n} P_i$ for nodes $N_1, .., N_n$.

Fig. 8. Heterogeneous cluster with 21 devices ranging from powerful x86 to weaker RISC-V and ARM devices used to evaluate Beanstalk.

Deterministic Program Optimization with Probe Set Partitioning Applications that only spawn threads and allocate memory at startup are common for embedded edge applications, resulting in deterministic behavior when coupled with Wasm's deterministic execution model. We can further optimize shared access detection for these cases further by partitioning our probe masks across the cluster to instrument disjoint subsets of accesses. Since Wasm uses structured `loop` instructions, our instrumentation tool can also easily distribute the overhead of instrumentation within loops equally across nodes (Fig. 7) to prevent any one node from incurring exorbitant overheads and minimizing both the worst-case slowdown and jitter. The final set in this case can be computed as $P_F = (\bigcup_{i=1}^{n} P_i) \cup (\bigcup_{\alpha:R[\alpha]} I_F[\alpha])$ where $R[\alpha] = S_F[\alpha] \vee (T_F[\alpha] > 1)$, $S_F[\alpha] = \bigvee_{i=1}^{n} S_i[\alpha]$, $I_F[\alpha] = \bigcup_{i=1}^{n} I_i[\alpha]$, and $T_F[\alpha] = \bigcup_{i=1}^{n} \{T_i[\alpha].tid\}$ for all addresses α. This formulation of P_F contains no false positives under determinism, since memory access interleavings observed on every node N_i can theoretically be observed by performing the same instruction sequence on any other node.

5 Evaluation

We evaluate the benefit of execution diversity with Beanstalk by running 8 data-race benchmarks on our 21-node heterogeneous cluster (Sec. 5.1). To demonstrate the sensitivity of data-race bugs to device and instrumentation density heterogeneity, we constructed a statistical model of our debugging technique, which we use to quantify this sensitivity as the *heisen factor* (Sec. 5.2). We provide a statistical analysis of the bugs found (Sec. 5.3), and show that *heisenbugs* are often the norm, not the exception. Finally, we demonstrate how Beanstalk can leverage device and instrumentation heterogeneity to find more bugs than a homogenous baseline for the same budget, or deploy to resource-constrained environments without sacrificing the ability to find data-race bugs (Sec. 5.4). To attribute the effects of our statistical analysis *purely* to execution diversity, we avoid probe-set pruning in our debugger's evaluation, which is an extra optimization usable in practice by large-scale deployments.

5.1 Evaluation Setup

Distributed Debugging Cluster We deploy Beanstalk on a cluster of 21 heterogeneous nodes (Fig. 8; see supplementary material for detailed hardware specifications) running the WAMR [15] WebAssembly engine in Linux (Ubuntu 22.04). Each Beanstalk Engine uses probe mask-based instrumentation to locally activate a *random subset* of probes. The probe masks are generated prior to each run by sampling a uniform random distribution across all possible subsets that activate a

specified proportion of probes; we refer to this proportion of activated probes as the *instrumentation density* δ. To minimize timing overhead and jitter, all experiments configure Beanstalk Engines to use a short $\mu = 50$ for data-race detection delay loops (Alg. 1). While increasing the delay generally increases the chance of observing bugs, we reap decreasing benefits with increasing the μ window size.

Benchmarks To study heisenbug properties accurately in a controlled setting, we rigorously analyze the execution of 8 Linux benchmarks (see supplementary material for detailed descriptions) compiled to Wasm against WALI [67], which include:

- 4 hand-crafted benchmarks with inserted data-races,
- the *DataRaceBench* [50] benchmark suite with 3 races from real applications, and
- the *liblfds* [2] lock-free data structure library with unknown potential races;

We select these small benchmarks in our controlled experiments to *maximize our sample size* and statistical power behind our claims supporting diversity-guided heisenbug detection. Each benchmark is also statically compiled against the *musl* C standard library which serves as an additional source of unknown potential races. Most off-the-shelf benchmark suites, including *DataRaceBench*, rely on OpenMP for multithreading, which is not supported in the Wasm compiler toolchain yet; we manually transform these benchmarks to model the compiled structure of OpenMP sections to the best of our ability. In practice, Beanstalk is usable unmodified on *any* Wasm program agnostic of their size or system interface (including Emscripten [5], WASI [9], or WASIX [10]).

Experiments Each application was repeatedly executed for 3 minutes on 21 devices in our cluster and 20 different instrumentation densities ($\delta \in \{5, 10, \ldots 100\}\%$), resulting in **2, 619, 387** total datapoints. We also ran a naive baseline with $\delta = 100\%$ for 30 minutes on each of 3 identical machines. The large sample size of our experiments allows us to run statistically realistic simulations.

5.2 The Debugging Model and Heisen Factor

We identify that the observability of bugs can be greatly impacted not just by instrumentation density but also by the device on which a program is run during debugging (Fig. 11). To analyze these effects mathematically, we define a statistical model for our debugger, which we use to define and estimate the *heisen factor* — a metric which quantifies the degree to which a program's execution environment affects the observability of a given bug.

Beanstalk Debugging Model Let parameters δ and V represent the instrumentation density and target device, respectively. For any given bug, let $B^{(t)}$ be the event that the debugger detects the bug and $M^{(t)}$ be the event that the probes required to detect said bug are activated for any run t of the program. We define the following terms:

- The *detectability* $Y = \mathbb{P}(B^{(t)})$ measures the *overall performance of the debugger*. This is the probability that a bug is reported by our debugger in a single program run (as observed by our collected dataset), which depends on the execution environment and $\{\delta, V\}$.
- The *observability* $X = \mathbb{P}(B^{(t)}|M^{(t)})$ measures the probability that a bug occurs when being debugged. This captures the *behavior of the bug* (agnostic of the debugger) and also depends on $\{\delta, V\}$.

For notational convenience, we let $\Phi = \mathbb{P}(M^{(t)})$. Since Beanstalk chooses a random δ proportion of probes to activate and bugs *must* be probed to be detectable (i.e, $\mathbb{P}(M^{(t)}|B^{(t)}) = 1$), it follows

that:

$$\mathbb{P}(M^{(t)}) = \begin{cases} \delta^2 & \text{if threads race on different instruction sites (two distinct probes)} \\ \delta & \text{if threads race on the same instruction site (only one probe)} \end{cases} \tag{1}$$

$$X = \frac{\mathbb{P}(M^{(t)}|B^{(t)}) \cdot \mathbb{P}(B^{(t)})}{\mathbb{P}(M^{(t)})} = \frac{\mathbb{P}(B^{(t)})}{\mathbb{P}(M^{(t)})} = \Phi^{-1} \cdot Y \tag{2}$$

Heisen Factor Definition We define the *heisen factor* F as the coefficient of variation of observability X across the susceptibility parameters $\{\delta, V\}$:

$$F = \frac{\mathrm{Var}_{\delta,V}(X)^{1/2}}{\mathbb{E}_{\delta,V}(X)}. \tag{3}$$

where *Var* and $\mathbb{E}$ represent the variance and mean respectively. When F is small, the observability X is largely constant, while a large F indicates a high variance in the observability under different susceptibility parameters. The heisen factor F is a positive real-valued intuitive measure of the "heisen-ness" of a bug since the coefficient of variation normalizes for the average observability of the bug. Note that this formulation is easily extensible to include new susceptibility parameters beyond just devices (V) and density (δ) in the future without changing the core statistical setup.

Estimating the Heisen Factor The number of detections K of a bug is related to individual bug detections $B^{(t)}$ and the number of runs N by[4]

$$K = \sum_{t=1}^{N} \mathbf{1}_{B^{(t)}}. \tag{4}$$

This allows us to model K conditionally as the binomial distribution on N trials,

$$K|N, Y, \Phi, V \sim \mathrm{Binomial}(N, \Phi X). \tag{5}$$

While a naive estimate of $\hat{X} = \frac{K}{N\Phi}$ is possible, this estimate is biased towards a large F for rare bugs[5], and requires correction; however, frequentist approaches are infeasible, since we have a very limited number of observations of rare bugs. We thus use a Bayesian approach with a $Beta(\alpha, \beta = 1)$ *prior* – a standard conjugate prior for a binomial, and configure α with a reasonable data-driven estimate of the mean such that $\mathbb{E}[Y|K, N, \Phi, V]$ matches the observed fraction $\hat{Y} = K/N$ to capture our unbiased prior belief about the overall rarity of bugs. Applying this prior yields the posterior:

$$Y|K, N, \Phi, V \sim \mathrm{Beta}(\alpha + K, \beta + N - K) \tag{6}$$

We then estimate the heisen factor F with a maximum-a-posteriori (MAP) estimator, treating each (Φ, V) as empirical samples of equal weight. While there is no closed-form expression for the sum (or sum of squares) of a combination of beta-distributed random variables, we can estimate the posterior distribution of F using a Monte-Carlo simulation by independently sampling each observability X from their respective beta distributions:

$$F|K, N = \frac{\mathrm{Var}_{\Phi,V}(X|K, N)^{1/2}}{\mathbb{E}_{\Phi,V}[X|K, N]} \tag{7}$$

[4] $\mathbf{1}_{B^{(t)}}$ denotes the indicator function of the event (set) $B^{(t)}$.

[5] This can be seen through a hypothetical bug with constant observability $X = \varepsilon$, which has $\mathrm{Var}(\hat{X}|\delta, V) \approx \frac{\varepsilon\Phi}{N}$ and $\mathbb{E}[\hat{X}|\delta, V] = \varepsilon\Phi$; even if δ is fixed, and all devices V are identical, as $\varepsilon \to 0$, $F \approx \frac{1}{\sqrt{N\Phi\varepsilon}} \longrightarrow \infty$.

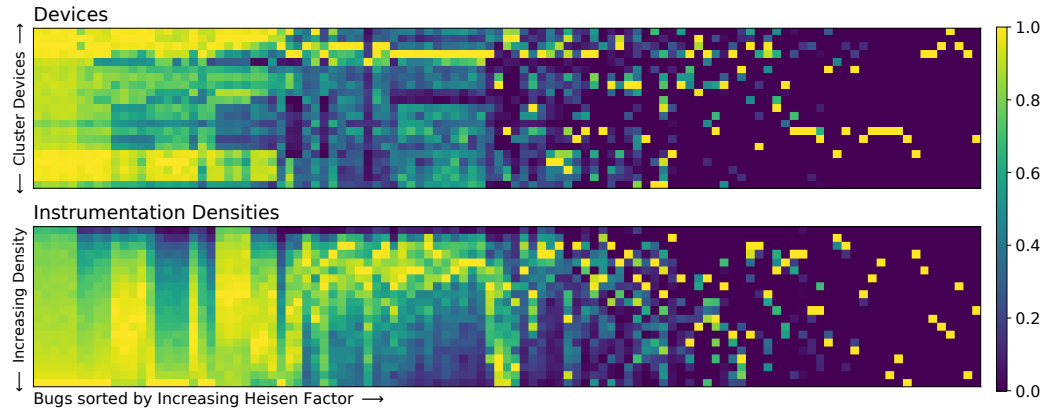

Fig. 9. Heat map of the observability of data-race bugs across different devices (averaging across instrumentation densities) and instrumentation densities (averaging across devices), sorted by estimated heisen factor F and normalized by the best observability. Bugs with small F have uniform observability across different devices and densities, while bugs with large F exhibit greatly varying behavior.

where $\text{Var}_{\Phi,V}$ and $\mathbb{E}_{\Phi,V}$ are taken as sums over the observed (Φ, V) samples[6]. Finally, the MAP is computed using a Gaussian kernel density estimate, which we report in our analyses below.

5.3 Beanstalk Bug Analysis

Using the heisen factor as a quantitative measure for evaluating data-race detectors, we apply this principle to our implementation of Beanstalk. Surprisingly, we find that Heisenbugs are far from rare, and are actually more common than "ordinary" bugs in our benchmarks. Furthermore, the increased observability of Heisenbugs at specific lower instrumentation densities can even compensate for the decreased probability of instrumenting a bug, resulting in higher detectability. This indicates a clear advantage for execution diversity in a debugger.

Prevalence of Heisenbugs While some bugs have relatively uniform behavior across different instrumentation densities and devices in our cluster, many of the 109 bugs in our dataset exhibit a high heisen factor with significant degree of variation (Fig. 9). This is likely because long-standing software like *libc* and *liblfds* have already detected and fixed ordinary bugs that are easily reproducible, leaving mostly elusive heisenbugs in the pool. Note that bugs with the largest heisen factor are only observable on specific devices and densities, highlighting the criticality of heterogeneity in both debugging densities and platforms to finding bugs. Consequently, these bugs are also *less consistently reproducible* across different execution environments, making debugging efforts challenging. While the heisen factor is computed on a real-valued spectrum, natural thresholds can be drawn at boundaries exhibit stark differences in observability characteristics.

Heisen Factor and Observability Fig. 10 shows the estimated heisen factor of data-race bugs against the *best* observability of each bug across devices and instrumentation densities. We observe a general negative correlation between the heisen factor and observability, which matches common intuition that rarer bugs have more strange behavior. However, note that not all bugs with a large heisen factor are rare: some large heisen factor bugs can still have significant conditional

[6]Note that since $0 \le X \le 1$, this also imposes the constraint that $0 \le Y \le \Phi$. Although we still approximate the posterior as a beta distribution in order to use a beta-distributed conjugate prior, we sample Y from a truncated beta distribution with the (rare) case that $Y > \Phi$ (i.e. $X > 1$) removed.

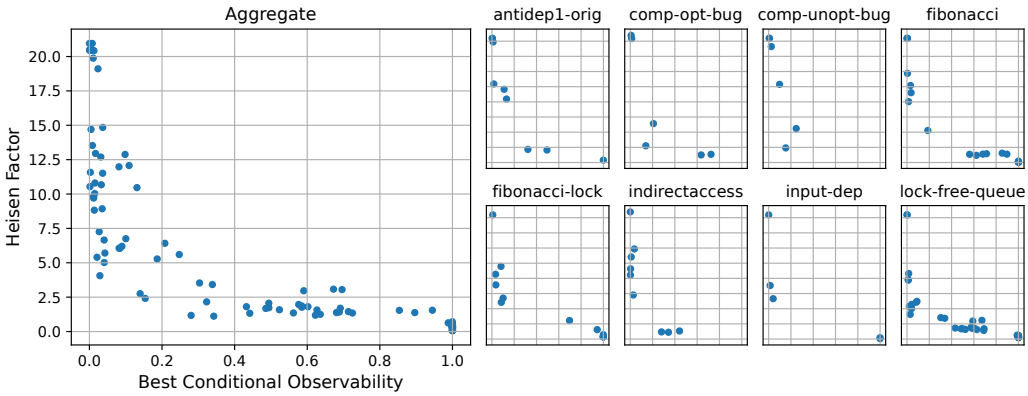

Fig. 10. Estimated heisen factor of data-race bugs plotted against the maximum observability of each bug across each device and instrumentation density for all benchmarks. Some benchmarks may only have few bugs (`input-dep`, `comp-unopt-bug`), but a large proportion of said bugs have a high heisen factor

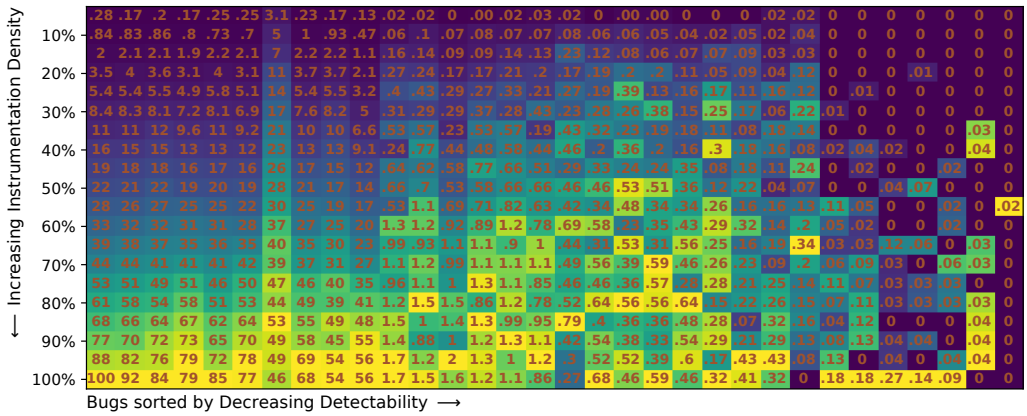

Fig. 11. Detectability of bugs found in the `lock-free-queue` benchmark with fixed device (Intel NUC 11, i7-1165G7) across instrumentation densities; each cell shows the detectability (in %), and is colored based on its detectability relative to the maximum detectability of each bug across all densities. While "ordinary bugs" are most detectable at maximum instrumentation densities, some "heisenbugs" are almost as detectable (or even more detectable) at specific, lower densities despite having fewer potential chances to catch those bugs.

observability. These bugs can be especially dangerous: testing in homogeneous environments is likely to miss them, while some deployments will encounter these bugs disproportionately, leading to potentially catastrophic outcomes *only in some environments*.

Increased observability can compensate for reduced instrumentation. Although lower instrumentation densities reduce the probability of instrumenting a bug on each run, the increased observability of heisenbugs at specific densities can result in a higher detectability (Fig. 11). This suggests that Beanstalk can potentially find many more bugs by running at varying instrumentation densities, or find a similar number of bugs even if Beanstalk is only allowed to run at low instrumentation densities. We observe a diverse profile of detectability patterns with Beanstalk (Fig. 12), including but not limited to the following:

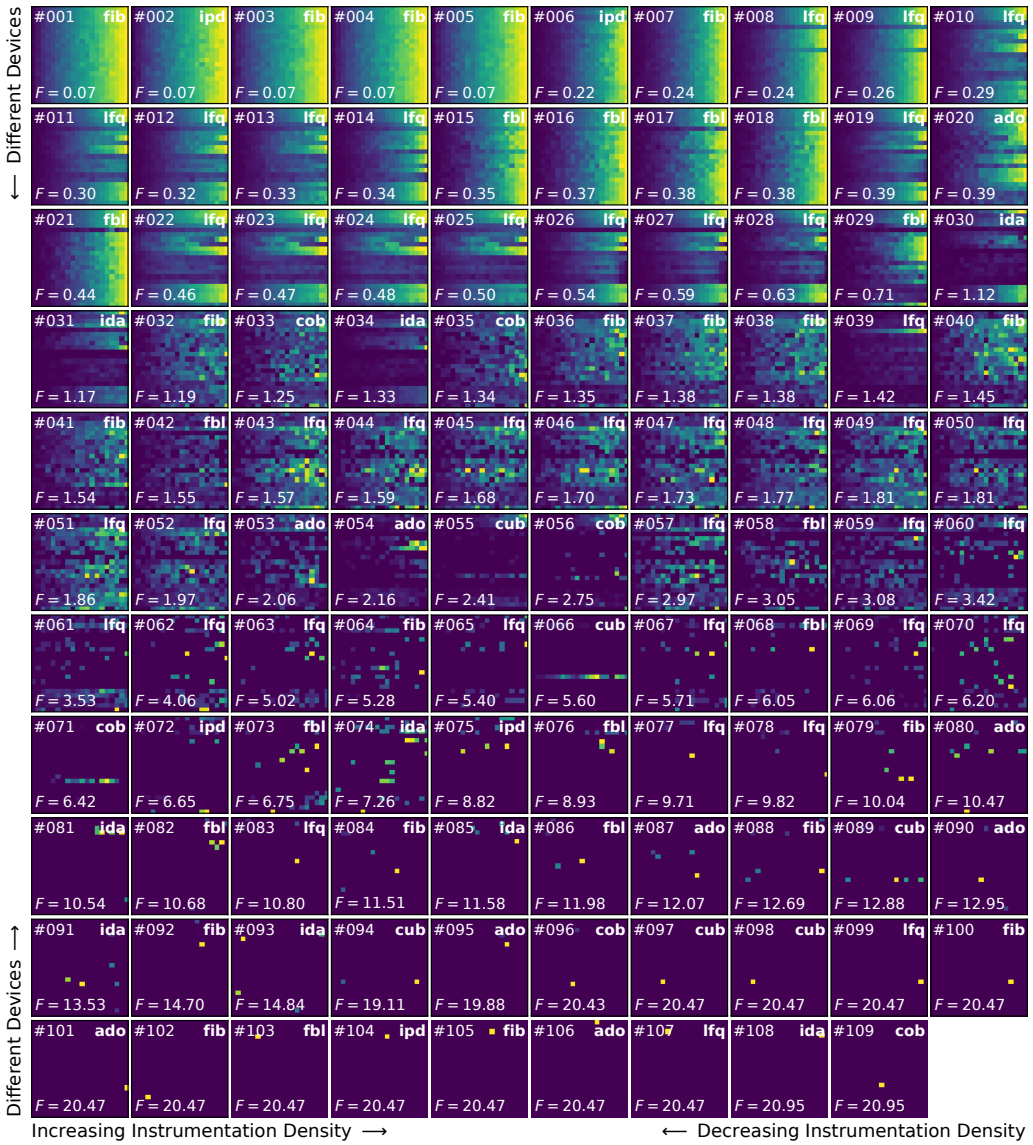

Fig. 12. Detectability profile of the 109 data-race bugs (benchmark acronym in top-right) found in our dataset sorted by the estimated heisen factor F. Each plot shows the detectability Y of a single bug across different devices (y-axis) and instrumentation densities (x-axis). Detectability of low F bugs increases with instrumentation density and are uniform across devices, but detectability of high F bugs are *overwhelmingly impacted* by observability at specific (often non-maximal) instrumentation densities and/or on specific devices.

- *Ordinary Bugs* (`#001-007`): These bugs are uniformly detectable across all devices and follow the prevalent notion that detectability is proportional to instrumentation density.
- *Device-Banded Bugs* (`#010-028,030,034,039,...`): These bugs exhibit a distinct band pattern (either low or high detectability) for four devices near the bottom, which correspond to

identical Intel TigerLake devices. This shows that identical devices are mostly redundant, providing negligible benefit for bug exploration.

- *Device-Specific Bugs* (`#039,066,071,081,089`): These bugs exhibit a particularly high detectability only on *one specific device* which differ on a bug-by-bug basis. Such bugs can be especially difficult to reproduce or debug if they are tested on the wrong device.
- *"Random" Bugs* (`#032,035,051,053,...`): These bugs lack a discernible pattern in detectability related to instrumentation or devices. This suggests that their behavior may either be completely random, or that other unaccounted susceptibility parameters may be required to explain this behavior.

We only notice these effects *statistically* — analytical inspection of the buggy configurations does not point to an obvious reason behind these patterns and similar devices do show divergent behavior often. We offer several suppositions as to why we may observe such statistical behaviors:

- *Unique Performance Regimes*: Some devices operate significantly slower (e.g. StarFive VF2) – more than an order of magnitude – compared to faster devices (e.g. NUC 11). This disparity is so pronounced that a minimally instrumented slower device often runs slower than a maximally instrumented faster device. Such performance diversity enables exploration of different operational regimes where bugs become more noticeable. This principle is also applicable to interpreted versus compiled execution of programs, which can be 20x slower.
- *Phase Changes*: Timing-influenced phase changes in programs, either explicitly programmed or implicitly driven by scheduling and queueing characteristics of the OS, can cause subtle instrumentation changes to trigger shifts into distinct phases with unique code coverage and bug patterns. For example, bug `#068` in `fibonacci-lock` only happens if thread 1 executes the start of its routine before the loop iteration for creating thread 2.

5.4 Debugging Performance

We evaluated the performance of Beanstalk by simulating 10000 runs on each benchmark, and compared Beanstalk against a "homogeneous" baseline which instruments every probe point and runs on homogeneous devices. Simulation was conducted by sampling collected runs from our cluster with an various compute budgets between 1 and 60 minutes, which was split uniformly at random between each instrumentation density and device in our cluster. We also consider Beanstalk's ability to operate in soft real-time deployments by evaluating it under constrained instrumentation density instead of a fixed compute budget.

Detection Probability Beanstalk is able to find bugs at least comparable to baselines not exploiting diversity, with many bugs being impossible to find without both device and instrumentation diversity. Fig. 13 shows the CDF of the detection probability of data-race bugs in our benchmarks, which can be broadly split in three regions:

(1) Easy-to-detect bugs that are always found ($\approx$ 50% of bugs are detected with probability 1.0);
(2) "Ordinary" but less-common bugs that are found at a similar rate regardless of system heterogeneity; and
(3) Rare heisenbugs only found in specific deployments ($\approx$ 30% of bugs are not detected at all without both device and instrumentation density heterogeneity.

Note that for higher detection probabilities (> 0.8), Beanstalk is marginally weaker than its competing ablations. This is logical since bugs that are most observable within a consistent configuration are better served by continuous testing in that same environment versus exploring different, potentially less observable configurations with Beanstalk. However, it is also largely inconsequential since less observable bugs are the most critical to effectively detect – highly observable bugs

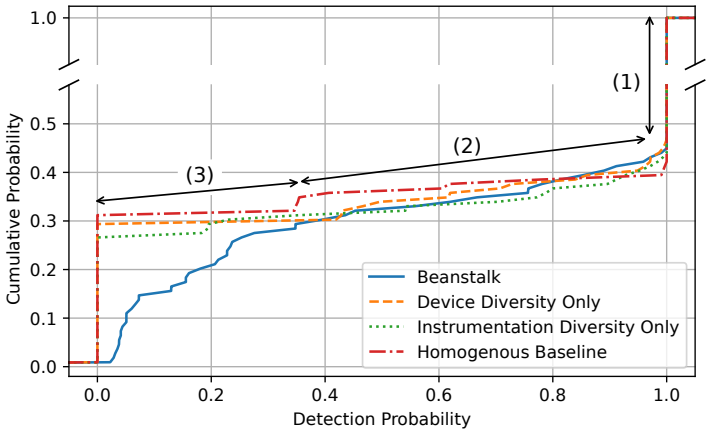

Fig. 13. Detection probability CDF of bugs in our benchmarks for a 30-minute compute budget (lower and to the right is better). Nearly 30% of bugs are undetectable without both instrumentation and platform diversity

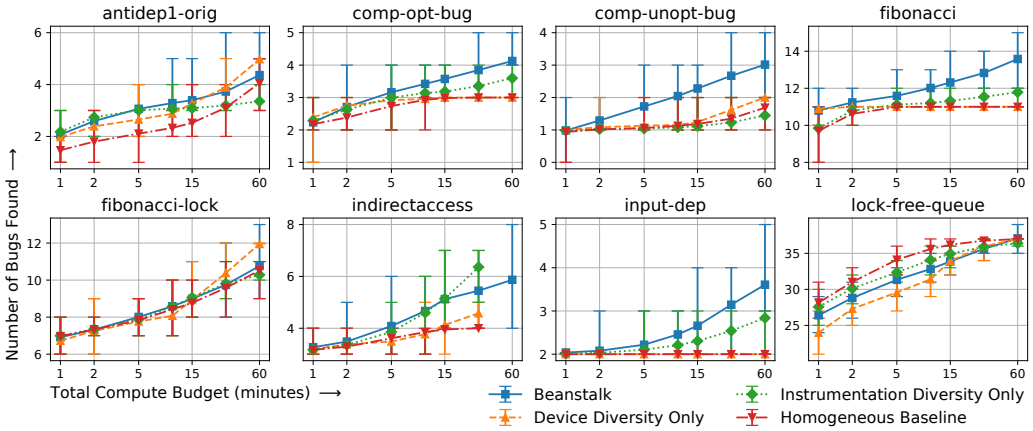

Fig. 14. Debugging campaign simulations of Beanstalk along with ablations showing the impact of device and instrumentation density heterogeneity on varying compute budgets showing the average, 5th, and 95th percentile of the number of bugs found averaged across 10000 runs. Beanstalk finds more bugs on average than the baseline, and in some cases even has a 5% worst-case performance which exceeds the 95% best-case performance of the baseline.

will likely be identified quickly during testing. Our instrumentation-diversity ablation subsumes the capability of traditional debugging methodologies used in TSVD or DataCollider, which means these tools, deployed as they are today, will completely miss region `(3)`.

Compute Budget To evaluate the overhead-effectiveness tradeoff of instrumentation and device heterogeneity for a given compute budget, we ran ablations on Beanstalk for each aspect by only using homogeneous devices and fixing the instrumentation density at 100% respectively. In our simulated debugging campaigns, Beanstalk is able to exploit device and instrumentation density heterogeneity to find at least as many bugs (on average) compared to these baselines (Fig. 14) at most evaluated compute budgets. Instrumentation diversity alone performs better than the baseline homogeneous devices in some cases (`comp-opt-bug`, `indirectaccess`), as does device

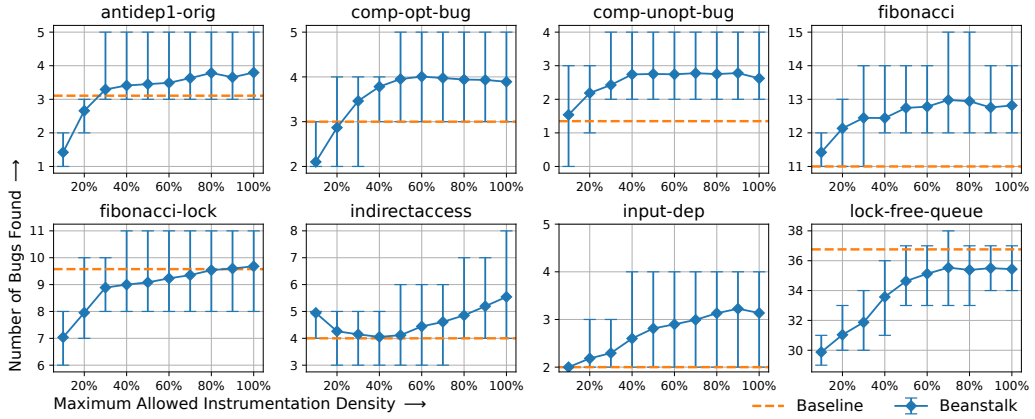

Fig. 15. Debugging simulations of Beanstalk with varying maximum allowed instrumentation densities (as a proxy for compute overhead constraints) with a 30-minute compute budget; the mean number of bugs found by the homogenous baseline is shown as a dotted line. While Beanstalk tends to find the most bugs when more instrumentation densities are available, Beanstalk finds more bugs than the baseline in many cases even when handicapped by overhead restrictions.

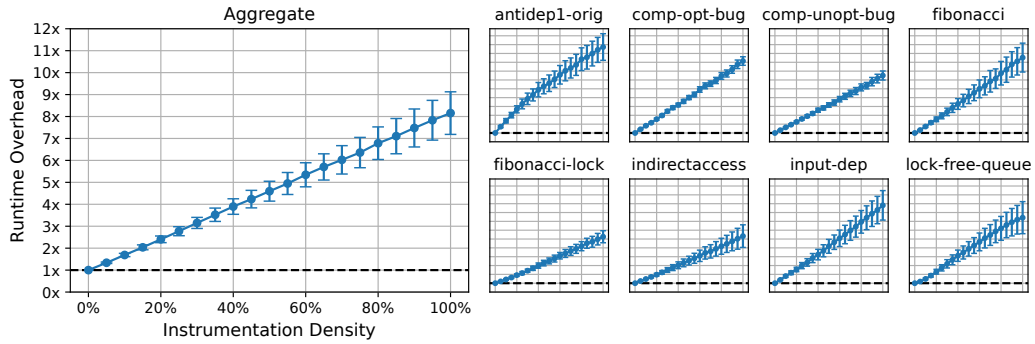

Fig. 16. Aggregate overhead of Beanstalk's data race detection instrumentation (geometric mean across benchmarks and ±2 standard errors across devices in our heterogeneous cluster) along with per-benchmark overhead breakdowns (with the same x-y scale). The average device runs 100% instrumented benchmarks up to 9x slower than uninstrumented execution, with some benchmarks up to 11x slower (`antidep1-orig`).

diversity alone (`fibonacci`, `antidep1-orig`). These however can even reduce the number of discovered bugs since redundant microarchitectural variation and slower devices respectively and make inefficient use of the same budget (`lock-free-queue`). By combining both device and instrumentation heterogeneity, Beanstalk creates a more holistic debugger, which can find more bugs than the baseline in almost all cases on average, with a consistently superior 5%/95% performance. A noteworthy trend is that Beanstalk achieves the best scalability with increasing compute budgets, likely because it minimizes the likelihood of running redundant execution interleavings with both device and instrumentation diversity.

Instrumentation Density Systems at deployment can usually tolerate a limited compute overhead, which can be heuristically translated to a maximum allowed instrumentation density $\delta_{\max}$. To

evaluate the impact of decreasing the allowable density, we simulated Beanstalk with a fixed budget of 30 minutes and varying $\delta_{\max} = \{10\%, 20\%, \ldots 100\%\}$ (Fig. 15). While low maximum densities tend to reduce the ability for Beanstalk to find bugs with a constrained runtime budget, Beanstalk can still often find nearly as many bugs while simultaneously operating at lower overheads (Fig. 16). The more devices Beanstalk scales across, the more powerful the detector becomes.

Note that this simulation does not account for the *scalability* advantage of running with lower overhead. Devices in a distributed edge system may have varying degrees of unused computing resources; with low instrumentation densities, Beanstalk can run on a wider range of devices and utilize otherwise unusable compute effectively without impacting overall system performance. Furthermore, while we simulate a fixed instrumentation density across all devices, powerful devices may be capable of more intensive instrumentation than weak devices at any given operating point.

6 Discussion and Implications

In this section, we provide concrete recommendations for the future of debugger design in light of Beanstalk to tackle heisenbugs in several contexts and propose an intuition for extending the heisen factor beyond data-race detection demonstrated here.

Debugging in Controlled Environments Optimizing for compute budget (Fig. 14) is paramount for in-house controlled debug farms: given a fixed cluster of devices and a fixed amount of time, the framework must best utilize its budget to filter out the most bugs. Using observed bug characteristics from simulations with Beanstalk, we conclude that selecting a *large number of diverse platforms* — notably including both slow and fast machines — are much more beneficial than just a homogeneous cluster of high-power server-scale machines, in terms of *both* financial cost and detection effectiveness. Using *only* the intended deployed platforms can be dangerous since low-observability bugs may not manifest during testing, but testing on other platforms can magnify the observability of these bugs. This style of debugging could even be offered as a cloud-service, with companies purchasing fixed compute budgets in well-selected heterogeneous environments that can utilize the whole spectrum of instrumentation densities.

Debugging in the Wild Debugging in the wild has a number of benefits over in-house controlled debugging, including but not limited to:

(1) *Exploration of Unpredictable Inputs*: Testing a wide range of program inputs, especially in cyber-physical systems is extremely difficult given the unpredictability of physical environments and sensor data; deployed systems often encounter more complex edge cases than in-house that lead to unprecedented bugs.
(2) *Operating at Scale*: Debugging can leverage several orders of magnitude larger scale than in-house, opening up opportunities for a net increased computational power and diversity.

However, unlike controlled environments where compute budget is the primary metric, minimizing runtime overhead is paramount for actively deployed systems to continue running without impacting normal operation. From our bounded-instrumentation simulations with Beanstalk (Fig. 15), we reason that operating effectively *is possible* even under minimal instrumentation with our diversity-guided methodology, often performing even better than baselines with just two dozen devices. Consider that typical edge deployments for IoT, industrial manufacturing, or autonomous systems involve orders of magnitude more devices, which can allow Beanstalk to scale seamlessly and scavenge heisenbugs at minuscule densities, especially when these systems inherently span diverse heterogeneous platforms. Our methodology can also be readily adapted to global ecosystems with less stringent requirements like Android/iOS app store, Web applications, or OS distributions.

To adopt this in practice, application developers would specify the expected compute usage of tasks and report any additional usable timing slack while maintaining satisfactory functionality.

Profiling across a quantized set of instrumentation densities (as in Fig. 16), they would select a cut-off instrumentation density up to which their application can operate within the slack, allowing for diversification up to that cutoff. Note that while higher cutoffs should ideally perform better than more strict cutoffs, our provided simulations for Beanstalk are not monotonically increasing. A larger cutoff produces more deployment configurations; since our simulations are averaged across randomly sampled configurations, the number of bugs found may decrease if increased instrumentation does not provide any better observability of bugs. This suggests that a more intelligent strategy can dynamically *learn* deployment configurations that will find more bugs than simple random sampling; we leave this as future work.

7 Related Work

Heisenbugs and Schedule Space Exploration Sallinger et al. [69] propose a formalization of heisenbugs in terms of a hyperproperty that relates sets of executions, treating the "heisen-ness" of a bug as a binary value (heisenbug or not-heisenbug) as opposed to our positive real-valued *heisen factor*. Scheduler-directed testing techniques [14, 25, 28, 37, 59], aimed at detecting and reproducing concurrency-related heisenbugs, is orthogonal to our diversity insight and could further enhance Beanstalk's search efficiency. PCT [14] lays the theoretical groundwork for probabilistic schedule-space exploration, inspiring subsequent efforts such as SKI [28] and Snowboard [32], which jointly explores test inputs and interleavings. CHESS [59] offers a comprehensive, scheduler-directed testing tool that incorporates HB analysis and addresses issues like starvation, frequent preemption, and imperfect replay. Racageddon [25] addresses efficient searching at runtime by using concolic execution and leveraging race-candidates for schedule improvement. More recently, machine learning has further refined these search techniques –Snowcat [33], for instance, employs a GNN-powered coverage predictor to uncover a number of elusive Linux kernel bugs.

Online Data-Race Detection Hardware assisted speedups using FPGAs [7], multicore parallelism [81], thread-local clocks [16], or performance monitoring units (PMUs) in processors [85] show promise for enabling online data-race detection at deployment, but are typically domain-specific. RaceFuzzer [72] uses low dynamic overhead and replayable analysis techniques but require hints about potential data-races from a prior analysis tool, making it complementary to our work. Racemob [42] explores scaling detection across multiple devices but does neither addresses heisenbug detection nor utilizes execution diversity, focusing instead on simulated data and a small homogeneous cluster for evaluation.

Data-Race Sanitization Automatically distinguishing between benign and harmful data races is important to reducing detection output, minimizing manual verification costs, and improving the usability of these tools for developers. Narayanasamy et al. propose identifying harmful races through differential execution testing across various race interleavings. In contrast, many tools [53, 72, 82] detect program exceptions or failures to identify such harmful races. Both approaches could integrate seamlessly with Beanstalk's established framework, leveraging parallel verification techniques from [42, 82] to more effectively post-process detection results. Triaging and aiding the repair of non-benign bugs (e.g. root-cause analysis) are beyond the scope of this paper, but future work could draw on techniques used in Portend [41],

Sampling and Statistical Debugging Efficient sampling and instrumentation of program locations is critical to low-overhead runtime analysis. Arnold et al. [8] propose a compiler-insert sampler with code duplication to enable tunable overheads at runtime, similar to Beanstalk, while reaping benefits from inlined code. [51] explores sampling in the context of bug isolation through refinement and statistical correlation, with [86] addressing multiple bugs simultaneously. HOLMES [18]

further highlights the benefits of path profiling for bug isolation. These techniques can be easily integrated into Beanstalk for bug isolation while maintaining low overhead with directed partial instrumentation and engine-supported code inlining (e.g. Wizard [4]).

Litmus Tests Stress testing memory-models using tiny microbenchmarks for referred to as *litmus tests* [54, 55] has been well explored. [30] uses device diversity to verify memory-consistency models of ARM64 with litmus tests. The concept of litmus tests could be adapted for developing standard heisenbug benchmark suites, consisting of a carefully selected set of known bugs that span a wide range of heisen factors.

8 Conclusion

Heisenbugs are a nightmare to debug, especially in cyber-physical edge systems with timing constraints deployed at scale that incorporate distributed hardware, platform heterogeneity, and operate in unpredictable physical environments. We demonstrate a debugging methodology harnessing *execution diversity* of both instrumentation and platforms, which is of vital importance to data-race heisenbug detection, with *Beanstalk*, an end-to-end automated and tunable data-race detection framework that uses sparse instrumentation, allowing easy integration in both debug farms and production deployment at scale. The methodology proposed is domain-agnostic, raising the exciting prospect of integration in IoT, automotive, mobile, OS distributions, or even Web ecosystems at a global scale. We envision that our statistical evaluation will also empower future debuggers – for bug classes even beyond data-races – to harness various forms of execution diversity and utilize our *heisen factor* metric accordingly to measure their ability to detect critical heisenbugs.

9 Acknowledgements

This work is supported by Bosch Research, the WebAssembly Research Center, and the National Science Foundation (NSF) grant awards: CNS-2148301, CNS-2148367, and TI-2229731. We also thank our anonymous reviewers for providing valuable feedback to improve the positioning and presentation quality of the paper.

10 Data-Availability Statement

All artifacts for the evaluation of Beanstalk, primarily the benchmarks, data collected on the cluster, and the simulation tools, are available on Zenodo [66]. The Beanstalk engine used for data collection on our cluster is also open-sourced on Github [75]. The cluster itself is internally managed, but the authors may be able to provide restricted access to it on an on-request basis.

References

[1] [n. d.]. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-2/. https://www.w3.org/TR/wasm-core-2/

[2] 2017. liblifds: A portable, license-free, lock-free data structure library written in C. https://liblfds.org/index.html. https://liblfds.org/index.html

[3] 2023. Software-related Recalls Are Still Plaguing the Auto Industry. https://www.sibros.tech/post/software-related-recalls-are-still-plaguing-the-auto-industry. https://www.sibros.tech/post/software-related-recalls-are-still-plaguing-the-auto-industry (Accessed 2023-1-31).

[4] 2023. Wizard Engine. https://github.com/titzer/wizard-engine. (Accessed 2023-04-11).

[5] 2024. Emscripten: An LLVM-to-WebAssembly Compiler. https://emscripten.org/. (Accessed 2024-10-02).

[6] Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. 2015. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 1–10. https://doi.org/10.1109/SIES.2015.7185039

[7] Faustin Ahishakiye, José Ignacio Requeno Jarabo, Violet Ka I Pun, and Volker Stolz. 2021. Hardware-assisted online data race detection. *Formal Methods in Outer Space: Essays Dedicated to Klaus Havelund on the Occasion of His 65th Birthday* (2021), 108–126. https://doi.org/10.1007/978-3-030-87348-6_6

[8] Matthew Arnold and Barbara G Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 168–179. https://doi.org/10.1145/378795.378832

[9] WASI authors. 2023. WASI: The WebAssembly System Interface. https://wasi.dev/. (Accessed 2023-8-08).

[10] Wasmer authors. 2023. WASIX: The Superset of WASI. https://wasix.org. (Accessed 2023-8-08).

[11] Automotive. 2023. Car software is riddled with bugs. https://privacyfirst.nl/en/articles/connected-cars-and-software-3-0-car-software-is-full-of-faults/

[12] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 81–96. https://doi.org/10.1145/1640089.1640096

[13] Eric Bodden and Klaus Havelund. 2010. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering* 36, 4 (2010), 509–527. https://doi.org/10.1109/TSE.2010.25

[14] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 167–178. https://doi.org/10.1145/1735970.1736040

[15] Bytecode Alliance. 2019. WebAssembly Micro Runtime. https://github.com/bytecodealliance/wasm-micro-runtime

[16] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A deployable sampling strategy for data race detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 810–821. https://doi.org/10.1145/2950290.2950310

[17] Rui Chen, Xiangying Guo, Yonghao Duan, Bin Gu, and Mengfei Yang. 2011. Static data race detection for interrupt-driven embedded software. In *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement-Companion*. IEEE, 47–52. https://doi.org/10.1109/SSIRI-C.2011.18

[18] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. 2009. Holmes: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 34–44. https://doi.org/10.1109/icse.2009.5070506

[19] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 258–269. https://doi.org/10.1145/512529.512560

[20] Domenico Cotroneo, Roberto Pietrantuono, Stefano Russo, and Kishor Trivedi. 2016. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. *Journal of Systems and Software* 113 (2016), 27–43. https://doi.org/10.1016/j.jss.2015.11.021

[21] Siwei Cui, Yifei Gao, Rainer Unterguggenberger, Wilfried Pichler, Sean Livingstone, and Jeff Huang. 2023. SmallRace: Static Race Detection for Dynamic Languages-A Case on Smalltalk. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1136–1147. https://doi.org/10.1109/ICSE48619.2023.00102

[22] Frank Denis. 2023. Performance of WebAssembly runtimes in 2023. https://00f.net/2023/01/04/webassembly-benchmark-2023/

[23] Christian El Salloum, Martin Elshuber, Oliver Höftberger, Haris Isakovic, and Armin Wasicek. 2013. The ACROSS MPSoC–A new generation of multi-core processors designed for safety–critical embedded systems. *Microprocessors and Microsystems* 37, 8 (2013), 1020–1032. https://doi.org/10.1016/j.micpro.2013.08.002

[24] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective {Data-Race} Detection for the Kernel. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. https://dl.acm.org/doi/10.5555/1924943.1924954

[25] Mahdi Eslamimehr and Jens Palsberg. 2014. Race directed scheduling of concurrent programs. *ACM SIGPLAN Notices* 49, 8 (2014), 301–314. https://doi.org/10.1145/2692916.2555263

[26] Kiko Fernandez-Reyes, Isaac Oscar Gariano, James Noble, Erin Greenwood-Thessman, Michael Homer, and Tobias Wrigstad. 2021. Dala: a simple capability-based dynamic language design for data race-freedom. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 1–17. https://doi.org/10.1145/3486607.3486747

[27] Cormac Flanagan and Stephen N Freund. 2010. FastTrack: efficient and precise dynamic race detection. *Commun. ACM* 53, 11 (2010), 93–101. https://doi.org/10.1145/1543135.1542490

[28] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. 2014. {SKI}: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 415–431. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/fonseca

[29] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 265–279. https://doi.org/10.1145/3423211.3425680

[30] Gallium. [n. d.]. Aarch64 Litmus Test. https://gallium.inria.fr/~maranget/cats7/model-aarch64/index.html.

[31] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, Chen, and Qi Alfred. 2020. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 385–396. https://doi.org/10.1145/3377811.3380397

[32] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 66–83. https://doi.org/10.1145/3477132.3483549

[33] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2023. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 35–51. https://doi.org/10.1145/3600006.3613148

[34] Ok-Kyoon Ha. 2013. Case Study of Dynamic Detectors for Data Races. *IERI Procedia* 4 (2013), 174–180. https://doi.org/10.1016/j.ieri.2013.11.025

[35] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200. https://doi.org/10.1145/3062341.3062363

[36] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 317–328. https://doi.org/10.1145/3238147.3238213

[37] Shiyou Huang, Bowen Cai, and Jeff Huang. 2017. Towards {Production-Run} Heisenbugs Reproduction on Commercial Hardware. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 403–415. https://www.usenix.org/conference/atc17/technical-sessions/presentation/huang

[38] Martin Jacobsson and Jonas Willén. 2018. Virtual machine execution for wearables based on WebAssembly. In *EAI International Conference on Body Area Networks*. Springer, 381–389. https://doi.org/10.1007/978-3-030-29897-5_33

[39] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[40] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering*. 13–22. https://doi.org/10.1145/1595696.1595701

[41] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data races vs. data race bugs: telling the difference with portend. *ACM SIGPLAN Notices* 47, 4 (2012), 185–198. https://doi.org/10.1145/2248487.2150997

[42] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced data race detection. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 406–422. https://doi.org/10.1145/2517349.2522736

[43] Steffen Keul. 2011. Tuning static data race analysis for automotive control software. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 45–54. https://doi.org/10.1109/SCAM.2011.16

[44] Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press. https://doc.rust-lang.org/book/

[45] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196. https://doi.org/10.1145/3335772.3335934

[46] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 323–337. https://doi.org/10.1145/161494.161501

[47] William Landi and Barbara G Ryder. 1992. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices* 27, 7 (1992), 235–248. https://doi.org/10.1145/143103.143137

[48] Chao Li, Rui Chen, Boxiang Wang, Zhixuan Wang, Tingting Yu, Yunsong Jiang, Bin Gu, and Mengfei Yang. 2023. An Empirical Study on Concurrency Bugs in Interrupt-Driven Embedded Software. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1345–1356. https://doi.org/10.1145/3597926.3598140

[49] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 162–180. https://doi.org/10.1145/3341301.3359638

[50] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1145/3126908.3126958

[51] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. 2003. Bug isolation via remote program sampling. *ACM Sigplan Notices* 38, 5 (2003), 141–154. https://doi.org/10.1145/780822.781148

[52] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When threads meet events: efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 725–739. https://doi.org/10.1145/3453483.3454073

[53] Kai Lu, Zhendong Wu, Xiaoping Wang, Chen Chen, and Xu Zhou. 2015. Racechecker: efficient identification of harmful data races. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 78–85. https://doi.org/10.1109/PDP.2015.19

[54] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated synthesis of comprehensive memory model litmus test suites. *ACM SIGPLAN Notices* 52, 4 (2017), 661–675. https://doi.org/10.1145/3093336.3037723

[55] Sela Mador-Haim, Rajeev Alur, and Milo MK Martin. 2011. Litmus tests for comparing memory consistency models: How long do they need to be?. In *Proceedings of the 48th Design Automation Conference*. 504–509. https://doi.org/10.1145/2024724.2024842

[56] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Petr Tma, and Zhengwei Qi. 2012. Java bytecode instrumentation made easy: The disl framework for dynamic program analysis. In *Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings 10*. Springer, 256–263. https://doi.org/10.1007/978-3-642-35182-2_18

[57] William P McCartney and Nigamanth Sridhar. 2006. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. 167–180. https://doi.org/10.1145/1182807.1182825

[58] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2022. WebAssembly as a Common Layer for the Cloud-edge Continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*. 3–8. https://doi.org/10.1145/3526059.3533618

[59] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs.. In *OSDI*, Vol. 8. https://www.usenix.org/conference/osdi-08/finding-and-reproducing-heisenbugs-concurrent-programs

[60] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319. https://doi.org/10.1145/1133981.1134018

[61] Manuel Nieke, Lennart Almstedt, and Rüdiger Kapitza. 2021. Edgedancer: Secure Mobile WebAssembly Services on the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3434770.3459731

[62] J Pancik, A Vemola, R Kledus, M Semela, and A Bradac. 2018. Auto recalls and software quality in the automotive sector. *EAI Endorsed Transactions on Scalable Information Systems* 5, 17 (2018). https://doi.org/10.4108/eai.29-5-2018.154808

[63] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 25–36. https://doi.org/10.1145/1508244.1508249

[64] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 327–340. https://doi.org/10.1002/cpe.1064

[65] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 1 (2011), 1–55. https://doi.org/10.1145/1889997.1890000

[66] Arjun Ramesh. 2025. *Unveiling Heisenbugs with Diversified Execution – Beanstalk Evaluation Artifact*. https://doi.org/10.5281/zenodo.14933663

[67] Arjun Ramesh, Tianshu Huang, Ben L Titzer, and Anthony Rowe. 2025. Empowering WebAssembly with Thin Kernel Interfaces. (2025). https://doi.org/10.1145/3689031.3717470

[68] Emily Ruppel. 2023. When Wasm Meets Cyber-Physical Systems: A Discussion of WebAssembly in Real-Time, Safety-Critical Systems. https://events.linuxfoundation.org/wasmcon/program/schedule/.

[69] Sarah Sallinger, Georg Weissenbacher, and Florian Zuleger. 2023. A formalization of heisenbugs and their causes. In *International Conference on Software Engineering and Formal Methods*. Springer, 282–300. https://doi.org/10.1007/978-3-031-47115-5_16

[70] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411. https://doi.org/10.1145/265924.265927

[71] Fabian Scheidl. 2020. WebAssembly: Paving the Way Towards a Unified and Distributed Intra-Vehicle Computing- and Data-Acquisition-Platform?. In *2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*. IEEE, 1–6. https://doi.org/10.23919/AEITAUTOMOTIVE50086.2020.9307384

[72] Koushik Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 11–21. https://doi.org/10.1145/2692916.2555263
[73] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71. https://doi.org/10.1145/1791194.1791203
[74] Kristin Shaw. 2024. The EV industry faces a core challenge: software bugs. https://www.popsci.com/technology/ev-software-bugs/
[75] Silverline Framework. 2023. Runtime Manager for Beanstalk Engine. https://github.com/SilverLineFramework/runtime-manager/tree/datarace
[76] David A Tomassi. 2018. Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 980–982. https://doi.org/10.1145/3236024.3275439
[77] Christoph Von Praun and Thomas R Gross. 2001. Object race detection. *Acm Sigplan Notices* 36, 11 (2001), 70–82. https://doi.org/10.1145/504311.504288
[78] Dolores R Wallace and D Richard Kuhn. 2001. Failure modes in medical device software: an analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering* 8, 04 (2001), 351–371. https://doi.org/10.1142/S021853930100058X
[79] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of WebAssembly for Embedded Systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 1–4. https://doi.org/10.1109/MECO55406.2022.9797106
[80] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring the edge up to speed with a webassembly os. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 353–360. https://doi.org/10.1109/PerComWorkshops48775.2020.9156135
[81] Benjamin Wester, David Devecsery, Peter M Chen, Jason Flinn, and Satish Narayanasamy. 2013. Parallelizing data race detection. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 27–38. https://doi.org/10.1145/2451116.2451120
[82] Zhendong Wu, Kai Lu, Xiaoping Wang, Xu Zhou, and Chen Chen. 2015. Detecting harmful data races through parallel verification. *The Journal of Supercomputing* 71 (2015), 2922–2943. https://doi.org/10.1007/s11227-015-1418-8
[83] Zachary Yedidia. 2024. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 649–665. https://doi.org/10.1145/3620665.3640408
[84] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 221–234. https://doi.org/10.1145/1095810.1095832
[85] Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. Prorace: Practical data race detection for production use. *ACM SIGPLAN Notices* 52, 4 (2017), 149–162. https://doi.org/10.1145/3093336.3037708
[86] Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*. 1105–1112. https://doi.org/10.1145/1143844.1143983

Received 2024-10-15; accepted 2025-02-18