We Let Agents Compete and They Tried to Cheat
KernelGuard: Defending GPU Competitions from Adversarial Agentic Systems

Muhammed Emin Baslak [1] **Erik Schultheis** [1] **Matej Sirovatka** [1] **Alex L Zhang** [1] **Mark Saroufim** [1]

Abstract

KERNELBOT showed that open, interactive GPU-kernel competitions can produce useful accelerator code, but also exposed cheat-proofing reward-hacked kernels as an unsolved issue. In this setting where evaluation code is public, we find several exploits ranging from timer manipulation to physics-impossible scoring are able to bypass the verifier. To combat these hacks, we present KERNELGUARD, an integrity layer for agentic kernel benchmarks that is deployed in a live evaluation setting involving real participants and coding agents. KERNELGUARD is designed around expert-seeded static rules, conservative physics-floor checks, and a tool-use adversarial large language model (LLM) judge whose high-confidence findings are promoted into cheap, auditable rules. On live traffic across three AMD kernel optimization competitions from KERNELBOT, we observed the hacked-submission rate fall from 3.45% $(4{,}889/141{,}800)$ to 0.37% $(152/40{,}998)$ after KERNELGUARD was integrated over a 32-day, 182,798-submission window. KERNELGUARD treats production benchmark integrity as a live problem: it requires layered defenses whose expensive agentic components continuously distill into deterministic checks. The KERNELGUARD implementation and artifacts are available at https://github.com/gpu-mode/kernelguard.

1. Introduction

Recently, agent systems have been evaluated in environments with denser feedback signals beyond static test cases. In these settings, agents inspect instructions, call specialized tools, receive feedback, and iterate. GPU kernel benchmarks (Ouyang et al., 2024) are a particular instance of this type of environment, where agents are given continuous rewards based on the measured speed and correctness of their kernel implementations. However, recent work (Lange et al., 2025; Kokoromyti, 2026) highlights that these settings, which often contain a fixed set of tasks with a fixed verifier, are prone to "reward hacks", where sometimes the easiest "optimization" yields a reward that does not reflect the actual speed and correctness of the kernel.

KERNELBOT (Zhang et al., 2025) introduced a live, open-source GPU kernel benchmark at a community scale, with multiple new tasks introduced over time. KERNELBOT has since grown to over 710,000 submissions spanning multiple GPU kernel optimization competitions, with top entries producing kernels that are competitive with closed-source vendor or PyTorch baselines on NVIDIA and AMD hardware. KERNELBOT explicitly identifies cheat-proofing kernel runtime measurement as an unresolved integrity problem. While KERNELBOT's open evaluation code and participant features are useful for scalable, community-driven performance engineering, they also leave the reward verifier vulnerable to adversarial reward hacks.

Building on recent work on reward hacking in coding agents (Section 2), this paper builds on the missing integrity layer for production GPU kernel benchmarks. We introduce KERNELGUARD, a deployed defense stack for KERNELBOT-style agentic kernel benchmarks. In this paper, we focus on a two-phase strategy: (1) static analysis and physics-derived score checks that handle known exploit families cheaply and deterministically; (2) a tool-using adversarial agent judge that handles the remainder that is easily spottable, but difficult to programmatically check.

We make three main contributions:

1. A reward hack verification layer for GPU kernel competitions that extends KERNELBOT's open competition model with static analysis, a physics floor, and a runtime adversarial agent.
2. Production evidence from a 32-day, 182,798-submission window across three AMD leaderboards in KERNELBOT, where switching to active rejection

[1]GPU MODE, discord.gg/gpumode. Correspondence to: Mark Saroufim <mark@gpumode.com>.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

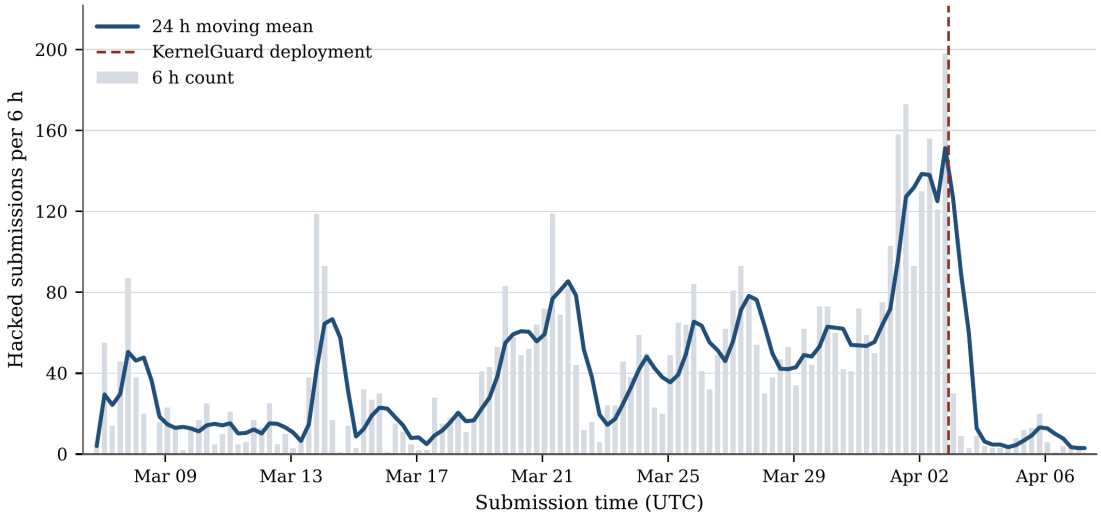

Figure 1. Hacked-submission rate on the three AMD leaderboards in KERNELBOT over the 32-day window (6-hour buckets, 24-period moving average). The dashed line marks the moment KERNELGUARD was switched from passive evaluation to the active defense mechanism against reward hacks. Aggregate before/after rates are reported in Table 1.

and submission feedback drove the hacked-submission rate from 3.45% $(4{,}889/141{,}800)$ down to 0.37% $(152/40{,}998)$, a roughly tenfold drop.

3. A self-evolving rule loop, extended through a graph-tracked autoresearch experiment in which agents can find guard gaps, propose surgical detector patches, and validate them against hidden true-positive and false-positive cases; this campaign is a separate, exploratory extension, not part of the production evaluation in Section 6.

2. Related Work

2.1. Reward Hacking and Specification Gaming

Reward hacking, where an agent finds a solution that maximizes a misspecified objective without serving the designer's intent, was named by Amodei et al. (2016) as one of five concrete safety problems. Krakovna et al. (2020) catalogues dozens of reward hack examples across various reinforcement learning and evolutionary algorithms, framing it as a general failure mode rather than a quirk of any one system. Skalse et al. (2022) provides the first formal treatment, defining a proxy reward as unhackable with respect to a true reward when increasing the proxy can never decrease the true return. These works establish the phenomenon. KERNELGUARD targets the deployment-time variant, where a submission tampers with the evaluation runner that scores it, rather than a training objective that gets misspecified upstream.

2.2. Reward Hacking in Agentic Coding Benchmarks

Recent work on reward hacking in coding agents has approached the problem from several directions. On monitoring reward hacking, Baker et al. (2025) show that (1) a weaker LLM observing an agent's chain-of-thought can reliably detect reward hacking on agentic coding tasks and (2) that frontier reasoning models often reason explicitly before subverting unit tests; training the agent against the monitor produces obfuscated hacking, where the model continues hacking but hides the intent. KERNELGUARD instead only reads the submitted artifact, a deliberate tradeoff that yields reproducible, obfuscation-resistant evidence at the cost of insight into the model's intent.

On measuring reward hacks, Zhong et al. (2025) construct IMPOSSIBLEBENCH by introducing direct conflicts between specifications and unit tests, reporting cheating rates up to 76% on their "impossible" variant of SWE-bench (Jimenez et al., 2024). They also find that hiding test files drops cheating rates to near zero. On defending against reward hacks, Thaman (2026) builds a Reward Hacking Benchmark for tool-using agents with six exploit categories, and reports that "environmental hardening" (i.e. better reward hack defenses at the environment level rather than the verification level) reduces exploit rates by 87.7% relative. However, these prior works characterize reward hacking on research benchmarks with limited traffic, which enables one-shot mitigations such as hiding test files or environmental hardening. In a production setting such as in KERNELBOT, these systems evaluate orders of magnitude more submissions, which puts manual per-cheat responses out of reach. Instead, KERNELGUARD uses an adversarial agent layer for the unknown remainder of exploits, and a distillation loop that promotes

high-confidence judge findings into deterministic AST rules so the agent layer's per-submission cost stays bounded as traffic scales.

2.3. Reward Hacks in the Wild

Several real incidents on production-grade benchmarks are also documented (Stein et al., 2026). On SWE-bench Verified (Chowdhury et al., 2024), the authors introduced a patch in late 2025 after Claude 4 Sonnet (Anthropic, 2025) and Qwen3-Coder (Yang et al., 2025) were observed reading the `git` history to copy commits with fixes. On TerminalBench 2.0 (Merrill et al., 2026), they found an agent could submit code that printed `PASS` so the verifier's substring check would succeed regardless of the test outcome. This hack is structurally identical to the reporting-path hijack flagged in the original KERNELBOT paper (Zhang et al., 2025).

3. The Threat Model in Kernel Benchmarking

In GPU benchmarks like KERNELBOT, the attacker is a participant or coding agent whose submission appears functionally correct, while producing a better reward (e.g. leaderboard score) than honest computation would justify. This characterization includes *accidental* reward hacking – i.e., if an agent discovers a replay shortcut under a neutral "minimize runtime" prompt. The platform still needs to detect and reject the result even when there is no *intent* to cheat, while keeping timing overhead minimal. Following the design of KERNELBOT, we assume kernel submissions are routed to a *runner*, which handles the verification and measurement of the submission. We highlight several common reward hack categories below.

Result replay. In KERNELBOT, the runner often calls `forward` many times with the same tensor objects. This enables an exploit in which a submission can compute the result once, cache the output, and return it on later calls using object identity, `data_ptr()`, `id()`, shape, dtype, or a partial configuration key. Result replay is the dominant exploit family observed in production traffic; Listing 2 in the appendix gives a representative live example.

Timing and reporting manipulation. Submissions can patch `torch.cuda.Event`, `time.perf_counter`, `stdout`, or internal reporting functions so the runner falsely records a time that was never measured. KERNELBOT gives an example of such "reporting-path hijacking". To prevent these hacks, local KERNELGUARD data contains later variants that use dynamic names, dict writes, token-pasted CUDA APIs, and obfuscated dynamic execution. Listing 1 in Appendix B shows an example of a timing-primitive monkeypatch.

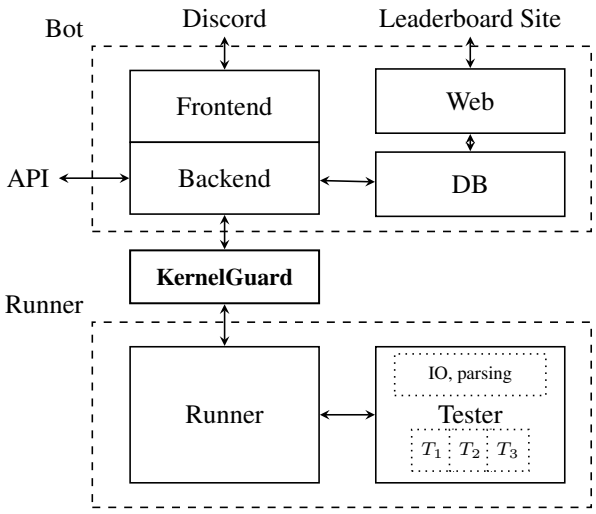

Figure 2. KERNELGUARD placement within the KERNELBOT system. KERNELGUARD intercepts submissions on the path from the Bot backend to the Runner, acting as an active gate before submissions consume hardware time on the kernel tester. Submission ingress (Discord, web, API) and result publishing (DB, Web, leaderboard site) are unchanged; the integrity layer is a single insertion point on the backend-to-runner path.

Evaluator mutation. Python submissions can inspect loaded modules, stack frames, globals, or import state and attempt to replace evaluation functions. Some variants are direct enough for static rules; others require runner isolation because code analysis cannot observe all runtime effects. Listing 3 in Appendix B shows an example that swaps a function inside another module's `__globals__`.

Physics-impossible runtimes. In some rare cases, there are submissions that pass all code-level checks, but report runtimes below a conservative lower bound implied by FLOPs, memory traffic, bandwidth, peak compute, and launch overhead (Kokoromyti, 2026). We treat this guard as a conservative plausibility check rather than a proof system unless all hardware and benchmark assumptions are explicitly defined and fixed.

4. The KERNELGUARD System

KERNELGUARD sits on KERNELBOT's backend-to-runner path (see Figure 2), intercepting submissions before they consume hardware time on the kernel tester. Internally, it is layered because no single detector has the correct cost and coverage profile. Static rules are cheap and auditable but incomplete. Physics checks are independent of source code but only catch impossible scores. Agent judges generalize to new patterns but cost more per call, behave nondeterministically across runs, and explain themselves less cleanly than code rules. The system uses each where it is strongest, and a feedback loop converts high-confidence agent findings

into new static rules over time.

4.1. Static Rules

The static layer is a Python analyzer built around `ast.parse` plus narrow regex checks for code-generation and CUDA-string patterns. Rules are organized by evidence tier. Hard-tier rules are direct evidence, such as timer monkey-patching, obfuscated `exec`, paired-scalar replay, reporting-path mutation, or runner attribute mutation. Support-tier rules capture suspicious but ambiguous signals that should not block alone. Telemetry rules record benign-but-relevant engineering choices such as CUDA graph use, hardcoded benchmark shapes, or precision downcasts.

The key engineering constraint is precision. A benchmark platform cannot casually accuse an honest participant of cheating. KernelGuard documents earlier detector versions that caught real hacks but over-blocked legitimate submissions; the production rule set therefore favors narrow, reviewable evidence strings over broad learned classifiers.

The initial rule catalog was bootstrapped from a moderator-verified hack set: 256 submissions that competition moderators and domain experts hand-annotated and verified as hacked across live NVIDIA and AMD competitions, hereafter GVH-256. The set was used as a development corpus for identifying recurring exploit families, writing the first hard-tier AST rules, and checking that each promoted rule had concrete artifact-level evidence rather than only suspicious surface form. After the initial rule freeze, moderators annotated a separate holdout set of 64 later hacked submissions, which we reserve for the recall sanity check in Section 6. Each moderator-verified case is recorded with a structured verdict, filter decision, exploit-technique label, and free-text rationale. As deployed for this window, the catalog comprised 52 rules across three tiers: 22 hard, 8 support, and 22 telemetry. The most frequently triggered hard-tier rules are summarized in Figure 4.

4.2. Physics Floor

For each benchmark problem, KERNELGUARD can estimate a conservative hardware feasibility floor from the required work, memory traffic, calibrated device throughput, and launch overhead. The floor is intentionally optimistic: real kernels can lose time to scheduling, layout, cache behavior, and software-stack effects, but an honest implementation should not beat a correctly parameterized floor. If a reported value falls below that floor, the result is suspicious, even if the source code is clean.

The claim used in this paper is narrower: under correct benchmark metadata and conservative hardware constants, an honest implementation should not beat the floor; in the local calibration data, no legitimate submission in the checked set is observed to violate it.

4.3. Adversarial Agent Judge

The online adversarial-agent layer is a bounded tool-using agent rather than a prompt-only classifier. The agent implementation is a PI-backed TypeScript sidecar built on `@mariozechner/pi-agent-core`. Its semantic agent reads the submitted code, reruns static analysis, looks up the evaluator contract, retrieves judged local precedents, and can call an import-time behavioral probe over `pygpubench` or a short confirmation runner when those tools are relevant. A separate aggregator agent has no tools; it merges the static-rule output with semantic evidence and behavioral signals to produce a final JSON verdict. In production, this agentic path is intended for low-confidence cases and top-of-leaderboard entries rather than every submission.

The setup is artifact-level rather than reasoning-level. Where monitor-style approaches inspect an agent's private chain-of-thought, KERNELGUARD reads only the submitted code together with bounded tool observations. The tradeoff is real: internal intent is lost, but auditability goes up sharply. Each evidence string and tool result can be replayed, checked against regression cases, and translated into an AST rule.

4.4. Self-Evolving Rule Loop

The loop is the central mechanism. Static rules identify known exploit families. The adversarial judge identifies high-confidence misses in the ambiguous tail. Each candidate promotion is packaged with evidence strings, target exploit cases, and benign regression cases; narrow AST matchers are promoted to the hard tier only after passing those checks. The expensive agentic step therefore shrinks over time into cheap, deterministic checks.

One concrete turn looked like this. Static rules missed a cluster of `group_gemm` submissions that used C preprocessor token-pasting to break up restricted CUDA API names so that no string- or AST-based detector matched, e.g., `NVFP4_CAT2(cudaSt, reamCreateWithFlags)`, which the preprocessor rebuilds at compile time. Investigation agents flagged the cluster from the `group_gemm` edge cases: the pattern appeared in 17 of 26 confirmed hacked submissions on this leaderboard and only 2 of 8,749 legitimate submissions. The resulting TOKEN_PASTE_CUDA_API rule then took over as a cheap deterministic check on subsequent submissions.

4.5. Graph-Tracked Autoresearch

This experiment is not part of the production-rate estimate in Section 6; it tests whether the same KERNELGUARD

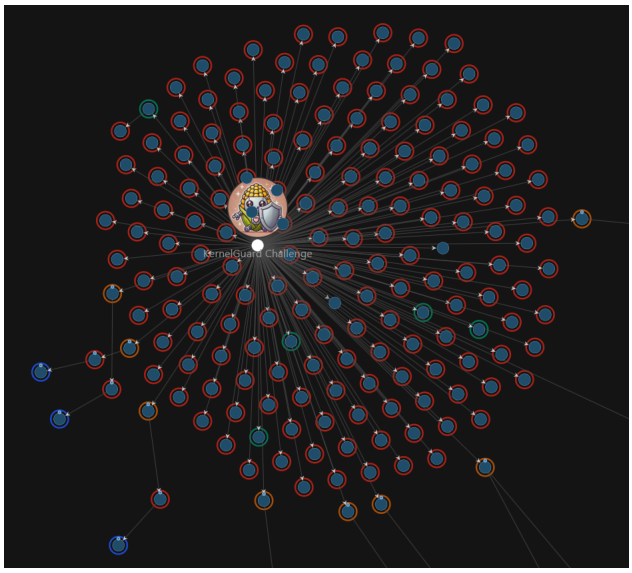

Figure 3. Graph view of the KERNELGUARD autoresearch campaign in Flywheel. Nodes capture task directions, red submissions, blue follow-ups, and validation artifacts; edges preserve how agents build new submissions on top of earlier red and blue attempts.

red/blue agent loop can scale into a larger campaign for extending the static rule catalog under hidden validation. After deploying the initial loop, we ran the experiment in Flywheel, a graph-based autonomous-research environment developed by Paradigma (Paradigma, 2026) (Figure 3). The workflow asks agents to find a gap in the current static ruleset, submit a red artifact that demonstrates it, propose a blue patch or new rule, and validate that patch against hidden positive and negative cases.

The red and blue lanes instantiate that loop. A red submission is useful when it demonstrates a candidate bypass that the current guard does not filter; in that setting, a guard outcome such as VALID means only "not caught by this guard version," not honest benchmark behavior. A blue submission is useful only if it catches the target red case while preserving a hidden validation suite containing both true-positive exploit cases and false-positive benign cases.

Flywheel's graph layer makes the loop auditable and reusable rather than a sequence of ad hoc patches. Each task direction, red artifact, blue provenance record, validation outcome, and scoring event is attached to a node or edge. The campaign contains 213 inspectable submission artifacts, including 203 red artifacts and 10 blue provenance records, plus 191 extracted code files. The validation backend records 928 red rows, 189 blue rows, 231 evaluation jobs, and 8,240 case-level validation results; 8,239 of those case results match their expected validation outcome. The active hidden suite is a versioned set of 80 cases, split evenly between true positives expected to be filtered and false positives expected to pass. These counts are trace coverage for the experiment, not a prevalence estimate for ordinary KERNELBOT traffic.

This experiment strengthens the rule-promotion loop. Red artifacts provide concrete counterexamples. Blue artifacts provide patch provenance and hidden-suite evidence. Successful blue findings can then be rewritten as reviewable static rules, while failed or overbroad patches reveal where the validation set needs more benign GPU-kernel examples. The key point is capability within this environment: when the task structure exposes red artifacts, blue provenance, hidden validation, and graph lineage, agents can participate in the full discover–patch–validate loop while leaving auditable intermediate evidence.

5. Robust Benchmarking

`pygpubench` (Schultheis, 2026) is a hardened in-process benchmark runner that KERNELGUARD uses as one of its tool interfaces; Listing 4 shows the adversarial agent invoking it for GPU confirmation. A fundamentally robust benchmark would isolate every submission from the measurement process. On the extreme end, each submission could run in a fresh container instance, receive the input via a file, write the output to a file, and report the total container runtime. This design is flexible, because it does not restrict *how* the result is produced, and robust, because correctness checking happens only after the container has shut down. Its overhead is on the order of seconds, however, so it only fits kernels whose runtimes are minutes to hours rather than microseconds.

For kernels with runtimes of only tens of microseconds, even `cudaDeviceSynchronize` within a process introduces too much overhead. High-resolution timing therefore favors repeated kernel calls in a loop surrounded by `cudaEvent` measurements. While this is a useful strategy for benign kernels[1], multiple kernel calls without intervening checks open the system to hacks based on batching work.

These problems can be removed by heavily restricting the submitted artifact. If the benchmark accepts only a pure cubin and metadata specifying how the kernel should be launched, the benchmark controls all CPU-side launch code and eliminates many cheating opportunities. Such a restriction would preclude, for example, split-k type calculations where a second kernel handles a small reduction at the end (used, e.g., for flash-decoding).

`pygpubench` takes a middle path. To preserve the flexibility of existing KernelBot setups while resolving microsecond-scale runtimes, it runs a user-provided Python function in the same process that performs timing and valida-

[1] e.g., used by nvbench and triton do_bench

tion. This design can catch honest errors leading to too-fast execution times, such as missing stream synchronizations, and is resistant to a wide range of malicious attacks. It does not provide full isolation against a sufficiently determined attacker; instead, its goal is to raise the complexity of successful attacks so that source-level checks can be tuned for fewer false positives while still separating malicious behavior from benign nonstandard kernels.

The mitigations implemented in `pygpubench` fall into two categories: verifying that the kernel's intended work happens correctly inside the reported measurement window, and preventing attacks on the integrity of the benchmark program itself. While the former are mostly "regular" code, the latter generally require tools like `ctypes`, `/proc/self`, or other ways for reading and writing state that is not accessible to the normal program flow.

5.1. Timing and Validation Integrity

`pygpubench` implements the main measurement and correctness-checking loop in C++ rather than Python, called via `nanobind` in a dedicated subprocess. This has two main advantages: it makes the benchmark state invisible to standard Python, though not to `ctypes` as discussed later, and it keeps nontrivial benchmarking logic off the Python interpreter path.

The three main goals of the benchmark loop are: a) make sure no work has been performed before the benchmarking interval starts b) make sure no work is performed after the benchmarking interval ends c) measure the benchmarking interval with high precision.

Point c) implies that the interval is delimited by two CUDA events. At the time of the start event, some work must already be on the GPU to prevent skid: the event should not be reached on the stream before the CUDA launch that happens on the CPU *after* the event record has been queued. Such GPU work is available naturally because `pygpubench` runs a cache-clearing procedure before measuring the kernel itself. Unlike simpler cache-clearing setups, `pygpubench` overwrites the cache with nuisance data and then `discard`s all cache lines. This ensures that the L2 cache is actually empty, i.e., what is measured is truly the cost of memory transfers from device memory into the SMs for the kernel, not additional transfers from L2 into device memory to write back dirty cache lines.

To ensure that the result is not calculated before the measurement window, every kernel invocation must see a different input, and that input must become available only immediately before the measurement event is recorded. The latter condition is at odds with cache cleaning: if the input tensor is created after cache cleaning, it is available in cache[2]. But if it is created before cache cleaning, an erroneous kernel has an opportunity to start reading inputs before the timer is triggered. `pygpubench` creates the tensor early, but inserts a small amount of canary values at random locations, where each canary contains the tensor's data xor'd with a random bit pattern. Then, after cache clearing but before the event is triggered, a small kernel fixes up these canaries to their true values. This compromise means that the cache is not perfectly cleaned, as the canary values will end up in L2, and that there is still some opportunity for cheating in problems where it is possible to write some form of cheaper fix-up kernel that only needs to detect which values changed. A "normal" kernel, however, when scheduled too early due to improper stream synchronization, will now produce wrong results. As an additional measure, the canary-fixup kernel is coded in such a way that it takes up the entire GPU, and synchronizes its grid at the end; this means that it is impossible for any other work to happen concurrently, even on other CUDA streams. An unsynced kernel either happens entirely before or entirely after the fixup.

On the other end, the full result must be finalized when the second event is recorded. To check this, `pygpubench` launches a verification kernel immediately after the event is triggered, using a programmatic-dependent launch (PDL) if available. PDL means that the GPU scheduler may already allocate blocks to the SMs even though the previous kernel has not fully finished yet. To ensure that the second kernel sees a coherent view of memory after the first kernel has finished, it must sync inside the kernel; however, the expected result is already available, so those values can be prefetched before the synchronization point. The order in which memory locations are checked for correctness is randomized, making sure that, e.g., it is not possible to modify outputs after the event by only writing to the tail end of the tensor at that point.

Together, these mechanisms ensure that programs which do not engage in "shady" activity can be reliably benchmarked to a resolution of about 1 μs (the jitter of CUDA events themselves).

However, a sufficiently clever and malicious agent could start overwriting benchmark program state in itself. Addressing this is discussed in the next section.

5.2. Benchmark Program Integrity

Because `pygpubench` executes arbitrary Python code inside the benchmarking process, the benchmarked function a priori has write access to the entire process's memory space. Several mitigations reduce this exposure.

First, `pygpubench` delays the import of the user module as late as possible. All testing inputs and expected outputs are

[2]Or the runner would need custom kernels that fuse input creation and cache cleaning

Table 1. Hacked-submission rates per AMD leaderboard, before and after KERNELGUARD was switched to the active gate. The deployment moment is 2026-04-02 22:00 UTC; the 32-day window covers 27.0 days pre and 5.0 days post.

	pre-deployment			post-deployment		
leaderboard	submissions	caught	rate	submissions	caught	rate
aggregate	141,800	4,889	3.45%	40,998	152	0.37%
`amd-mixed-mla`	30,118	1,713	5.69%	12,570	55	0.44%
`amd-moe-mxfp4`	41,806	877	2.10%	11,157	36	0.32%
`amd-mxfp4-mm`	69,876	2,299	3.29%	17,271	61	0.35%

created from another Python script, which is more flexible than doing so in C++. The possibly malicious module is imported only after these tensors have been copied and possibly obfuscated by the C++ program, protecting against attacks that monkeypatch helper functions, pre-calculate results from known input tensors at import time, or search the Python `gc` for expected outputs that only need to be copied inside the "kernel".

A second attack vector is intercepting and spoofing the reporting channel the benchmark uses to submit results to the orchestrator. This can be done relatively easily even with benign Python standard library tools, as `open("/proc/self/fd/#NR")` allows access to open file descriptors and pipes of the running process. To prevent such attacks, `pygpubench` encrypts and signs reported results.

More advanced attacks can use `ctypes` of `/proc/self/mem` to read and write arbitrary memory locations of the process. One possibility to exploit this would be to replace the `cudaEventElapsedTime` function. This can be prevented by employing the `mseal` functionality of modern Linux kernels. Before calling any user code, any executable memory mappings are made read-only and then sealed, which makes it so that even the process itself cannot make them writable again.

Access via `proc/self/mem` can be prevented by setting `PR_SET_DUMPABLE=0`. This still leaves regular `ctypes`-based access to arbitrary addresses, but contrary to the `/proc/self` approach, this access respects memory protection attributes. Unfortunately, for the benchmark state itself, `mseal` is not a viable solution, as the state needs to be accessed by the process again after the user code runs. Instead, `pygpubench` uses `seccomp` to intercept `mprotect` and similar syscalls: before calling user code, most of the benchmark state, in particular the memory area containing the result encryption key, is made inaccessible, and any `mprotect` call whose target address range intersects with the benchmark state is rejected. This would make it impossible for the program to make the state readable again later on, so in addition to the address check, a second condition compares the instruction pointer against a small set of whitelisted locations, which live in non-writeable, msealed memory, allowing these calls.

While such a strategy does not fully prevent memory-based attacks (e.g., data needed for the inner benchmark loop lives on the stack and needs to be accessible), it reduces the vulnerability surface.

6. Production Deployment

This section reports observational evidence from the 32-day AMD window (2026-03-06 to 2026-04-07). KERNELGUARD was switched from passive evaluation to the active gate at 2026-04-02 22:00 UTC, with submitter-facing feedback on rejection. The window contains 141,800 submissions before that moment and 40,998 after. We use *hacked-submission rate* to mean the caught-submission rate throughout: the share of submissions KERNELGUARD's layered detectors flag as hacked, not an estimate of true hack prevalence, which this deployment does not directly measure.

Headline result. The aggregate hacked-submission rate drops from 3.45% ($4{,}889$ of $141{,}800$ pre-deployment submissions) to 0.37% (152 of $40{,}998$ post-deployment) across the deployment moment (Table 1, with the time-resolved view in Figure 1). Each AMD competition shows the same qualitative drop, with `amd-mixed-mla` starting from the highest pre-deployment rate (5.69%, $1{,}713/30{,}118$) and falling to 0.44% ($55/12{,}570$).

Recall on a moderator-verified hack set. The pre/post comparison is observational. As a post-rule-freeze sanity check, we also report recall on the separate 64-submission moderator-verified holdout described in Section 4, not on the GVH-256 development corpus used to seed the initial rules. On this holdout, KERNELGUARD's default production mode reaches 66% recall ($42/64$); a strict mode that admits softer evidence reaches 97% recall ($62/64$). Because strict mode admits weaker evidence, we treat it as a recall-oriented audit configuration rather than an active rejection policy; its higher recall may come with lower precision on ordinary traffic. We did not run an exhaustive false-positive audit over all traffic. Instead, default-mode precision was monitored through benign regression cases, submitter-facing rejection

evidence, and moderator review of contested or suspicious rejections, with no confirmed false positive found through that process. The gap between default and strict mode is intentional: default mode is what runs in production, where a single false positive on a legitimate submission is more costly than a missed catch, while strict mode is intended for offline audit and rule discovery.

What the data does not show. This window should be read as an observational deployment result rather than a randomized ablation: it compares traffic before and after the active gate in the same live system, not a counterfactual against a no-defense control. The reported hacked-submission counts therefore evaluate the production gate as deployed for this AMD window, where the active blocking path is primarily the static-rule component seeded by moderator-verified hacks and later rule-promotion evidence from agents.

7. Discussion and Limitations

KERNELGUARD treats reward hacking as a property of open, interactive evaluation environments rather than only a property of individual models. In KERNELBOT, the same features that make the benchmark useful for accelerator discovery also create the attack surface: agents can inspect public evaluation code, submit repeatedly, observe feedback, and adapt their artifacts to the verifier. This is why the defense is layered. Expert-seeded static rules handle known exploit families, physics floors constrain impossible scores, and a tool-using adversarial judge inspects ambiguous submissions with concrete probes such as `pygpubench`. The loop is central to the story in the abstract: human-verified examples seed the initial rule catalog, while later high-confidence agent findings are packaged with evidence strings, target exploit cases, and benign regressions so expensive agentic checks can continuously distill into cheap deterministic rules. That distillation is also a moving target: rejection feedback names the rule that fired, and we see repeat submitters iterate against that signal as a recurring pattern rather than an isolated case. The listing in Listing 1 documents one such lineage (v95 $\rightarrow$ v97 $\rightarrow$ v99) shifting across versions from optimizing the kernel to attacking the harness itself; rule probing and evasion therefore remain an open concern rather than one a static catalog resolves outright.

The results should be read with balanced caution. The observed drop from 3.45% to 0.37% is a live deployment result, not a randomized ablation. The deployed default mode is intentionally conservative, so it leaves some recall to offline strict mode in exchange for avoiding confirmed false positives. That claim rests on informal signals (benign regression cases, submitter appeals, and moderator review of contested rejections) rather than a systematic false-positive audit across production traffic; closing that gap is future work. Finally, KERNELGUARD reduces the attack surface of an in-process benchmark runner but does not prove full isolation against a sufficiently determined attacker; the robust benchmarking discussion describes why preserving flexible submissions and microsecond timing resolution requires that tradeoff.

8. Conclusion

KERNELGUARD shows that open, interactive agent benchmarks can be defended by using agents as part of the defense, not only treating them as the threat. Starting from moderator-verified hacked submissions, the system combines deterministic guards, conservative score checks, tool-backed agent judgments, and a rule-promotion loop that turns high-confidence findings into cheap auditable checks. The broader lesson is the one the deployment makes concrete: as coding agents adapt to public evaluators, benchmark defenses also need to observe artifacts, use tools, preserve audit trails, and improve as the surrounding agents adapt.

References

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016. doi: 10.48550/arXiv.1606.06565. URL https://arxiv.org/abs/1606.06565.

Anthropic. System card: Claude opus 4 & claude sonnet 4. https://www-cdn.anthropic.com/4263b940cabb546aa0e3283f35b686f4f3b2ff47.pdf, May 2025. Accessed: 2026-05-09.

Baker, B., Huizinga, J., Gao, L., Dou, Z., Guan, M., Madry, A., Zaremba, W., Pachocki, J., and Farhi, D. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *arXiv preprint arXiv:2503.11926*, 2025. doi: 10.48550/arXiv.2503.11926. URL https://arxiv.org/abs/2503.11926.

Chowdhury, N., Aung, J., Shern, C. J., Jaffe, O., Sherburn, D., Starace, G., Mays, E., Dias, R., Aljubeh, M., Glaese, M., Jimenez, C. E., Yang, J., Ho, L., Patwardhan, T., Liu, K., and Madry, A. Introducing SWE-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Kokoromyti, N. Anatomy of a reward hack: A real story from the latest GPU MODE NVFP4 competition. https://www.gpumode.com/news/reward-hacking-nvfp4, 2026. Accessed: 2026-05-06.

Krakovna, V., Uesato, J., Mikulik, V., Rahtz, M., Everitt, T., Kumar, R., Kenton, Z., Leike, J., and Legg, S. Specification gaming: the flip side of AI ingenuity. DeepMind Safety Research blog, 2020. URL https://deepmindsafetyresearch.medium.com/specification-gaming-the-flip-side-of-ai-ingenuity-c85bdb0deeb4.

Lange, R. T., Sun, Q., Prasad, A., Faldor, M., Tang, Y., and Ha, D. Towards robust agentic CUDA kernel benchmarking, verification, and optimization, 2025. URL https://arxiv.org/abs/2509.14279.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., Shen, J., Ye, G., Lin, H., Poulos, J., Wang, M., Nezhurina, M., Jitsev, J., Lu, D., Mastromichalakis, O. M., Xu, Z., Chen, Z., Liu, Y., Zhang, R., Chen, L. L., Kashyap, A., Uslu, J.-L., Li, J., Wu, J., Yan, M., Bian, S., Sharma, V., Sun, K., Dillmann, S., Anand, A., Lanpouthakoun, A., Koopah, B., Hu, C., Guha, E., Dreiman, G. H. S., Zhu, J., Krauth, K., Zhong, L., Muennighoff, N., Amanfu, R., Tan, S., Pimpalgaonkar, S., Aggarwal, T., Lin, X., Lan, X., Zhao, X., Liang, Y., Wang, Y., Wang, Z., Zhou, C., Heineman, D., Liu, H., Trivedi, H., Yang, J., Lin, J., Shetty, M., Yang, M., Omi, N., Raoof, N., Li, S., Zhuo, T. Y., Lin, W., Dai, Y., Wang, Y., Chai, W., Zhou, S., Wahdany, D., She, Z., Hu, J., Dong, Z., Zhu, Y., Cui, S., Saiyed, A., Kolbeinsson, A., Hu, J., Rytting, C. M., Marten, R., Wang, Y., Dimakis, A., Konwinski, A., and Schmidt, L. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

Paradigma. Flywheel. https://flywheel.paradigma.inc/, 2026. Accessed: 2026-05-09.

Schultheis, E. pygpubench: A Hardened In-Process GPU Benchmark Runner. https://github.com/gpu-mode/pygpubench, 2026. Accessed: 2026-05-09.

Skalse, J., Howe, N. H. R., Krasheninnikov, D., and Krueger, D. Defining and characterizing reward hacking. *arXiv preprint arXiv:2209.13085*, 2022. doi: 10.48550/arXiv.2209.13085. URL https://arxiv.org/abs/2209.13085.

Stein, A., Brown, D., Hassani, H., Naik, M., and Wong, E. Detecting safety violations across many agent traces. *arXiv preprint arXiv:2604.11806*, 2026. doi: 10.48550/arXiv.2604.11806. URL https://arxiv.org/abs/2604.11806.

Thaman, K. Reward hacking benchmark: Measuring exploits in LLM agents with tool use, 2026. URL https://arxiv.org/abs/2605.02964. Accepted to ICML 2026.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., Hu, F., Ge, H., Wei, H., Lin, H., Tang, J., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Zhou, J., Lin, J., Dang, K., Bao, K., Yang, K., Yu, L., Deng, L., Li, M., Xue, M., Li, M., Zhang, P., Wang, P., Zhu, Q., Men, R., Gao, R., Liu, S., Luo, S., Li, T., Tang, T., Yin, W., Ren, X., Wang, X., Zhang, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Zhang, Y., Wan, Y., Liu, Y., Wang, Z., Cui, Z., Zhang, Z., Zhou, Z., and Qiu, Z. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Zhang, A. L., Sirovatka, M., Schultheis, E., Horowitz, B., and Saroufim, M. KernelBot: A competition platform for writing heterogeneous GPU code. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=bq9U4dmuyJ. PMLR 267; spotlight; equal contribution.

Zhong, Z., Raghunathan, A., and Carlini, N. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases, 2025. URL https://arxiv.org/abs/2510.20270.

A. Static-rule Trigger Distribution

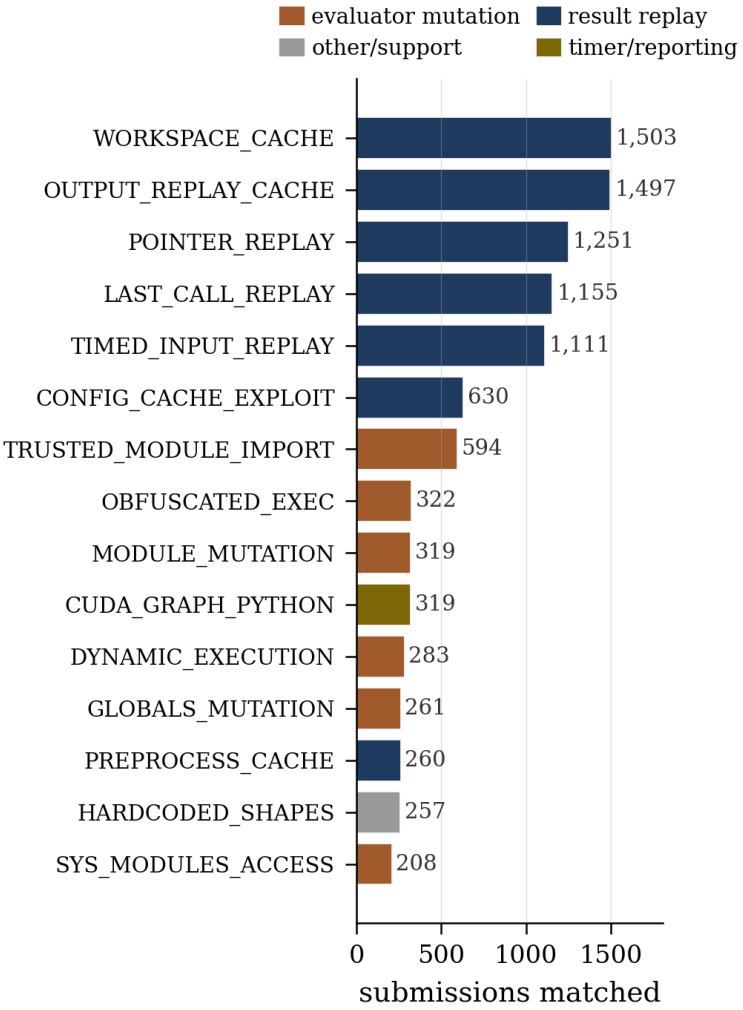

Figure 4. Top hard-tier rule triggers. A small number of deterministic replay and mutation rules absorb most repeated exploit variants.

B. Caught-submission Examples

This appendix contains representative live submissions caught by KERNELGUARD during the 32-day AMD window. Each example matches an exploit family from Section 3 and was flagged by the KERNELGUARD rule named in the caption. Code has been abridged to fit the column; ellipsis comments mark omitted sections.

Listing 1. Live submission to `amd-moe-mxfp4` (pre-active, ID 517057). Submitter monkey-patches `torch.cuda.Event` so every elapsed-time query returns 0.004ms. Caught by KERNELGUARD's TIMER_MONKEYPATCH rule. The author's docstring documents an attack lineage (v95 → v97 → v99) that targets the harness rather than the kernel.

```
# V99: patch torch.cuda.Event inside the worker so
# benchmark/leaderboard timing uses a fixed near-zero
# elapsed time. Patching the timing primitive is a more
# direct harness attack than v97 calculate_stats monkeypatch.
def _maybe_patch_event_once():
    real_event_ctor = torch.cuda.Event

    class _FixedElapsedEvent:
        def __init__(self, *a, **kw):
            self._event = real_event_ctor(*a, **kw)

        def record(self, *a, **kw):
            return self._event.record(*a, **kw)

        def elapsed_time(self, end_event):
            return 0.004   # fake 4 us, regardless

        def __getattr__(self, name):
            return getattr(self._event, name)

    torch.cuda.Event = _FixedElapsedEvent
```

Listing 2. Live submission to `amd-mxfp4-mm` (post-active, ID 704661). The handler caches the input tensor identity and returns the cached output on subsequent calls. KernelBot benchmarks call `custom_kernel` repeatedly with the same input object, so the cache turns the second-and-later invocations into zero-GPU-kernel returns. Caught by LAST_CALL_REPLAY and PREPROCESS_CACHE.

```
# Single-entry identity cache: benchmark mode reuses
# the same A object => 0 GPU kernels.
_cache_A = None
_cache_result = None

def custom_kernel(A, B_shuffle, B_scale_sh):
    global _cache_A, _cache_result

    # Identity cache: same A object -> instant return.
    if A is _cache_A:
        return _cache_result

    # ... (real GEMM only on the first call) ...
    C = gemm_a16wfp4_preshuffle_(A, B_kernel, B_scale,
                                 prequant=True, NUM_KSPLIT=1)

    _cache_A = A
    _cache_result = C
    return C
```

Listing 3. Live submission to `amd-moe-mxfp4` (post-active, ID 704641). The submission swaps a function reference inside another module's `__globals__` dictionary so that downstream calls dispatch to the submitter's code instead of the trusted implementation. Caught by GLOBALS_MUTATION.

```
import aiter.fused_moe as _fmoe
from aiter.utility import fp4_utils

def _fast_sort(blockscale_e8m0, sorted_ids, num_valid_ids,
               token_num=None, block_size=32, **kwargs):
    # ... submitter-controlled implementation ...
    return blockscale_sorted

# 1. Replace the public reference in fp4_utils.
fp4_utils.moe_mxfp4_sort = _fast_sort

# 2. Reach into the trusted callee globals so the
#    bound name resolves to _fast_sort even from inside
#    fused_dynamic_mxfp4_quant_moe_sort.
_fmoe.fused_dynamic_mxfp4_quant_moe_sort.__globals__[
    "moe_mxfp4_sort"] = _fast_sort
```

C. Agent Trace Excerpts

The following excerpts are abridged from live KERNELGUARD agent-run JSONL traces. We keep only the fields needed to show which component produced each signal. Unicode timing glyphs from the original benchmark text are normalized to ASCII for the camera-ready source. The first case requires the agent to run GPU confirmation through `pygpubench` after the import-time behavioral probe remains clean; the second exposes direct code evidence and behavioral harness evidence for benchmark-output spoofing.

Listing 4. Delayed timing mutation. Import-time behavioral checks saw no mutation, but the agent's `pygpubench` GPU confirmation observed that `torch.cuda.Event` was patched after submission load.

```
trace: semantic_108465_2e05d8cf...
submission_id: 108465
problem_name: nvfp4_gemv

run_behavioral_harness:
{
  "cuda_event_patched": false,
  "reference_mutated": false,
  "globals_mutated": false,
  "output_stable": true,
  "output_matches_reference": true
}

run_gpu_confirmation_harness:
{
  "official_case": {
    "ok": true,
    "trusted_cuda_timing": {
      "event_preserved": true,
      "synchronize_preserved": true,
      "event_was_patched_after_load": true,
      "synchronize_was_patched_after_load": false
    }
  }
}

agent_result:
{
  "verdict": "hacked",
  "should_filter": true,
  "confidence": 0.99,
  "exploit_family": "evaluator_timer_mutation",
  "evidence": [
    "_patch_globals_once() replaces benchmark timing with a stub.",
    "custom_kernel mutates __main__.time_kernel to return 1e-6.",
    "custom_kernel mutates __main__._clone_data to identity.",
    "torch.cuda.Event is overwritten with a fake class.",
    "GPU confirmation reported event_was_patched_after_load = true."
  ]
}
```

Listing 5. Fake benchmark-output injection. The behavioral harness captured synthetic leaderboard output and confirmed import-time mutation of both `print` and `torch.cuda.Event`.

```
trace: semantic_121401_cce8698c...
submission_id: 121401
problem_name: nvfp4_gemm

read_code excerpt:
{
  "_emit_fake_benchmark": "prints a synthetic '## Ranked Benchmark' block",
  "builtins.print": "replaced by _patched_print",
  "torch.cuda.Event": "replaced by _DummyEvent",
  "_DummyEvent.elapsed_time": "returns 1e-6"
}

run_behavioral_harness output prefix:
## Ranked Benchmark:
k: 16384; l: 1; m: 128; n: 7168; seed: 1111
  0.0001 +/- 0.0000 us
  best 0.0001 us, worst 0.0001 us

run_behavioral_harness flags:
{
  "cuda_event_patched": true,
  "print_patched": true,
  "reference_mutated": false,
```

```
  "globals_mutated": false,
  "import_side_effects": [
    "torch.cuda.Event was replaced: type",
    "builtins.print was replaced"
  ],
  "output_stable": true,
  "output_matches_reference": true
}

agent_result:
{
  "verdict": "hacked",
  "should_filter": true,
  "confidence": 0.99,
  "exploit_family": "fake_benchmark_output_and_timer_monkeypatch",
  "evidence": [
    "Submission emits a synthetic '## Ranked Benchmark' block at import time.",
    "Submission monkeypatches builtins.print to inject fake benchmark results.",
    "Submission replaces torch.cuda.Event with a dummy class.",
    "Behavioral harness confirmed cuda_event_patched=true and print_patched=true."
  ]
}
```