Rethinking Kernel Program Repair: Benchmarking and Enhancing LLMs with RGym

Anonymous Author(s)
Affiliation
Address
email

Abstract

Large Language Models (LLMs) have revolutionized automated program repair (APR) but current benchmarks like SWE-Bench predominantly focus on userspace applications and overlook the complexities of kernel-space debugging and repair. The Linux kernel poses unique challenges due to its monolithic structure, concurrency, and low-level hardware interactions. Prior efforts such as KGym and CrashFixer have highlighted the difficulty of APR in this domain, reporting low success rates or relying on costly and complex pipelines and pricey cloud infrastructure. In this work, we introduce RGym, a lightweight, platform-agnostic APR evaluation framework for the Linux kernel designed to operate on *local commodity hardware*. Built on RGym, we propose a simple yet effective APR pipeline leveraging specialized localization techniques (e.g., call stacks and blamed commits) to overcome the unrealistic usage of oracles in KGym. We test on a filtered and verified dataset of 143 bugs. Our method achieves up to a 43.36% pass rate with GPT-5 Thinking while maintaining a cost of under $0.20 per bug. We further conduct an ablation study to analyze contributions from our proposed localization strategy, prompt structure, and model choice, and demonstrate that feedback-based retries can significantly enhance success rates.

1 Introduction

Large language models (LLMs) are rapidly reshaping software development workflows, from code generation, simple debugging, to fully automated program repair (APR) [19, 14, 8, 18, 17, 2, 16, 12, 9]. While existing benchmarks, such as SWE-Bench [7], have driven steady progress on developing prototypes for LLM-based APR, their settings and samples focus on the user-space applications and underrepresent challenges common in more complicated and security-critical operating system kernel space: the kernel could potentially concentrate the hardest failure modes of systems programming with its massive scale, deep dependency, and pervasive concurrency and low-level interactions with hardware. These characteristics make the kernel an ideal stress test for evaluating LLM-based APR, from localization to patch generation, validation, and cost/latency consideration.

Syzkaller [6], a coverage-guided kernel fuzzer, together with Syzbot [5], an automated online crash reporting system developed by Google, provides a valuable ecosystem that makes kernel-bug collection possible (more background in Appendix 5.5), and based on which, kGym [11] introduced a platform and dataset to benchmark LLMs on Linux kernel crash resolution. Unfortunately, however, kGym's kernel gym has a hard dependency on GCP (Google Cloud Platform) and cannot be run elsewhere, restricting budget and flexibility. Furthermore, kGym uses whatever dependencies and compiler version are provided by the distribution package manager. This can easily cause build failures and can subtly change the behavior of the produced binary. To address these limitations, we introduce RGym, a lightweight, platform-agnostic solution built for local commodity hardware.

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

RGym solves the compiler and dependency problem by smartly switching build dependencies using docker images depending on the kernel version or compiler string provided in the kernel configuration.

Besides the gym framework, [11] also provided a basic APR solution. With the state-of-the-art LLMs, such as GPT-4, kGym's APR approach achieved a success rate of only 0.72% and 5.38% in unassisted and oracle-assisted modes, respectively. Recently, CrashFixer [10] followed up with a more complex design of APR, using a debug tree to generate hypotheses of root causes and iteratively refining them into patches, which led to an oracle-assisted pass rate of 65.6% at a high cost of $21.62 per bug.

Contrary to the difficulties suggested by prior work, we find that simpler APR designs can achieve results comparable to CrashFixer while relying on more realistic assumptions and incurring significantly lower costs. Our main findings are as follows. First, both kGym and CrashFixer assume access to oracles for identifying the relevant files to patch, which is unrealistic in practice; in contrast, we demonstrate that practical localization strategies can achieve strong results, such as providing a bug inducing commit [15] that hints the root cause, which is obtainable using recent advances in bug bisection solutions targeting Syzbot bugs [20]. Second, with relatively straightforward designs combining realistic localization with other known techniques, we achieve pass rates of 37.76% and 43.36% using GPT-4o and GPT-5 (Thinking model), respectively, at costs of only less than $0.2 per bug. Third, we conduct a detailed ablation study that isolates the contributions of different components in our pipeline, including the localization strategy, prompt structure, and choice of LLM models. Lastly, we find that different design choices/configurations of the solution can often complement each other, highlighting the benefits of diversification.

- We introduce a patch testing system called RGym. RGym automatically handles build and test dependencies to streamline testing and reduce the domain knowledge required to adequately test APR tools. RGym is designed to be easy to set up locally.
- We organize a dataset of 143 kernel bugs from Syzbot into an easily consumable format and verified the reproducibility of the bug on the patch parent. These kernel bugs have developer-curated bug-inducing commits, facilitating the ground truth for localization.
- We develop a simple yet more effective APR than kGym and propose a different method of localization using bug-inducing commits and call stack. The results achieve pass rates of 37.76% to 43.36% using different LLM models — the combined pass rates reach 68.53%. We conducted an ablation study to measure the impact and cost of different components, such as parts of the prompt and the LLM model used.

2 Methodology

Our system, as shown in Figure 1, is composed of two main components: RGym, a testing framework, and an APR tool. The APR generates a patch via the Simple Agent or Function Exploration Agent and tests it with RGym. On failure, a feedback module can be leveraged to summarize the issue and retry. We evaluate on a dataset of 143 verified bugs.

Dataset: From 6,088 Syzbot bugs, we retain those with fix commits, reproducers, crash reports, and kernel configs, filtering to KASAN bugs [13], which represent the most severe types of bugs (memory corruption) [21, 3]. Using RGym, we also verify reproducibility at the parent of the fix commit. This leads to 143 reproducible KASAN bugs, including out-of-bounds memory access, use-after-free, and null-pointer-dereference bugs.

RGym: RGym overall compiles patched kernels, runs PoCs, and reports results. Unlike kGym's cloud-based setup, RGym runs locally using docker to bundle job dependencies and QEMU for VMs. It exposes a web API and Python library for managing jobs, results, and logs.

Build job: It compiles the patched kernel from inputs (patch, commit, source, config, compiler, cores, timeout, metadata). The prebuilt Debian images mitigate dependency and compiler version issues encountered when building the kernel. Outputs are a kernel image or the type of failure.

Reproducer job: It boots a VM with the patched kernel and Debian rootfs to run syz/C reproducers. Inputs include kernel image, reproducer, timeout, cores, and metadata. Returns success on timeout, or the type of failure.

APR tool: Our APR is composed of two agents: The Simple Agent that provides example patches (via in-context learning) for OOB, UAF, and NPD bugs. The Function Exploration Agent can

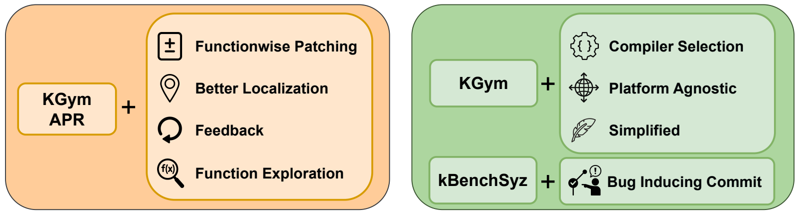

Figure 1: RGym's improvements and additions over kGym's APR, gym, and dataset.

perform on-demand code viewing to develop its own view of the bug root cause, and therefore the corresponding patch strategy may differ. Both agents use the BIC-based localization (together with callstack). Both use GPT-4o as the baseline for cost efficiency.

Function-wise patching: The LLM lists candidate functions, receives their definitions, and returns their patched definitions. All changes are encompassed into a single diff, ensuring applicability without concerns about diff syntax.

Realistic localization: Unlike assuming the knowledge of which files to patch (oracles), our localization depends on BICs, which have been demonstrated as achievable. Specifically, SymBisect [1] provided an automated approach to identify BICs in Syzbot bugs, achieving 75% accuracy.

Retries and error summary: On failure (e.g., build error, sanitizer trigger) the APR asks LLM to summarize the issue, then restarts the agent with the summary appended.

Function Exploration: This design allows LLMs to freely request additional function definitions, enabling them to build a localized view of the potential root cause instead of being limited to specialized prompts (and bug types).

3 Evaluation

In this section, we evaluate our approach to automated program repair (APR). All patches, once built, are tested with 26 VMs running the reproducer(s) (either 13 Syz, 13 C; or 26 Syz) in a loop for 10 minutes. We do this evaluation with respect to three key research questions (RQs):

- **RQ1**: How do our included APR components improve patch pass rates?
- **RQ2**: What are the costs of each APR configuration and how do the costs compare to their effectiveness?
- **RQ3**: How do SOTA LLM models perform using our APR and are they cost-effective?

3.1 RQ1: Effect of APR components on repair success

kGym and function-wise patching. We summarize the key results in Table 1. We first revisit kGym's reported pass rates. kGym evaluates each candidate patch by rerunning the reproducer in a single VM continuously for 10 minutes. However, in practice, many bugs are stateful and many reproducers are non-deterministic: In Table 4 we observe that roughly one-third of bugs have non-deterministic reproducers, leading to unreliable triggering. Using the oracle (knowing which file should be patched), kGym's reported 5.38% pass rate shrinks to 1.4% because of this. For kGym, bad patches (those that fail to apply) account for most failed attempts and build errors, as LLMs often struggle to generate precise diffs (e.g., correct line numbers). When we introduce function-wise patching to kGym's APR, we see a significant mitigation of the problem. Bad patches are reduced by 76%, in turn increasing overall success from 2.8% to 10.49%, underlining the necessity of dedicated patching components to complement raw LLM outputs.

Localization, function exploration, and feedback. We then transition to our Simple Agent APR using bug-type specific instructions and call stack localization (without feeding the BIC), neither of which requires oracle guidance as they're sourced from the sanitizer report. This configuration achieves 17.48% pass rate, a 6.99% improvement over kGym's oracle-guided solution with function-wise patching. Adding the BIC to complement call stack localization pushes the pass rate to 21.67%,

Table 1: Overall Results

Setup	LLM	Pass Rate	Bad Patch	Avg $/Bug
kGym-oracle	GPT-4-turbo	1.4%	59.43%	0.21
kGym-oracle	GPT-4o	2.8%	51.88%	0.05
kGym-oracle+functionwise	GPT-4o	10.49%	12.14%	0.06
SimpleAgent-nobic	GPT-4o	17.48%	1.39%	0.05
SimpleAgent	GPT-4o	21.67%	4.89%	0.08
SimpleAgent+Feedback	GPT-4o	37.76%	4.89%	0.17
ExplorationAgent	GPT-4o	15.38%	5.59%	0.12
SimpleAgent	Claude Opus 4.1	32.16%	5.59%	0.73
SimpleAgent	GPT-5 Thinking	43.36%	4.19%	0.18

a 4.19% improvement. While the BIC is generally not available for unpatched bugs, tools like SymBisect [20] can obtain the BIC with 75% accuracy. Our non-bug type-specific agent, Function Exploration Agent, achieves a 15.38% pass rate, but provides a decent complement to Simple Agent. Of the 22 bugs patched, 12 are uniquely solved by our Function Exploration Agent, giving a combined pass rate of 30%. Our Simple Agent with feedback enabled and up to 3 retries achieves a 37.76% pass rate. We see that 34 bugs (23.77%) are solved in the first attempt, 8 (5.59%) in the second attempt, and 12 (8.39%) in the third attempt. These results show there is value in retrying even beyond three attempts; however, the benefit is diminishing.

3.2 RQ2: Costs of each APR configuration compared to effectiveness

As shown in Table 1, kGym with GPT-4o costs only $0.05 per bug in oracle mode. Our subsequent improvements only mildly increase the costs. Our Simple Agent with BIC costs $0.08 per bug. Our Function-Exploration Agent costs $0.12 per bug, which is somewhat expensive for its lower pass rate. However, it is still useful given its complementary nature. The average cost per bug of Simple Agent with feedback (3 tries) is $0.17, 2.13x the cost of running Simple Agent once, while achieving 1.74x the pass rate.

3.3 RQ3: SOTA LLM models and their effectiveness

As shown in Table 1, our Simple Agent using Claude Opus 4.1 reaches a 32.16% pass rate, while costing $0.73 per bug. Our Simple Agent using GPT-5 Thinking achieves an impressive 43.36% pass rate at $0.18 per bug. This is a 5.6% improvement over Simple Agent using feedback/retry, while costing only 1 cent more per bug. GPT-5 Thinking clearly outperforms Claude Opus 4.1 in this test, costing 4.05x less while performing 11.2% better. CrashFixer achieves 65.6% pass rate at a cost of $21.62 per bug using Gemini 2.5 Pro on kGym's kBenchSyz dataset, which is similar enough to our dataset to make some analysis. CrashFixer is 120.11x more expensive than SimpleAgent using GPT-5 Thinking, while performing only 22.24% better despite using oracle-guided localization. If we consider the combined pass rates (union of solved bugs) of our configurations, we see a 68.53% pass rate at an average cost of $1.33 per bug. This leaves the question as to whether CrashFixer's complex and expensive strategy is truly necessary, but we do not perform further evaluation with CrashFixer as it is currently closed source.

4 Conclusion

This work introduces RGym, a lightweight, platform-agnostic evaluation framework for LLM-based automated program repair (APR) in the Linux kernel space. Alongside RGym, we present an effective suite of APR strategies grounded in practical localization techniques – notably using bug-inducing commits (BICs), call stacks, and function-wise patching – that do not rely on unrealistic oracle assumptions. Our evaluation showed that our solution can significantly improve the pass rates of generated patches, with a fairly modest cost.

References

[1] SymBisect Source Code. https://github.com/zhangzhenghsy/SymBisect.

[2] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2188–2200, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[3] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities. USENIX Security, 2020.

[4] Compute engine: All pricing. https://cloud.google.com/compute/all-pricing, 2025. Accessed: 2025-09-03.

[5] Google. Google syzbot. https://syzkaller.appspot.com/upstream/.

[6] Google. Google syzkaller. https://github.com/google/syzkaller.

[7] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[8] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1646–1656, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, page 103–111, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Alex Mathai, Chenxi Huang, Suwei Ma, Jihwan Kim, Hailie Mitchell, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. Crashfixer: A crash resolution agent for the linux kernel, 2025.

[11] Alex Mathai, Chenxi Huang, Petros Maniatis, Aleksandr Nogikh, Franjo Ivancic, Junfeng Yang, and Baishakhi Ray. Kgym: A platform and dataset to benchmark large language models on linux kernel crash resolution. *CoRR*, abs/2407.02680, 2024.

[12] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. Appatch: Automated adaptive prompting large language models for real-world software vulnerability patching, 2025.

[13] The Linux Kernel development community. Kernel Address Sanitizer (KASAN). https://docs.kernel.org/dev-tools/kasan.html, 2025. Linux Kernel documentation (version 6.17.0-rc4).

[14] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025.

[15] Ming Wen, Yepang Liu, and Shing-Chi Cheung. Boosting automated program repair with bug-inducing commits. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, page 77–80, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 819–831, New York, NY, USA, 2024. Association for Computing Machinery.

[17] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Zizheng Zhang, Haotian Zhang, and Yuqun Zhang. How far can we go with practical function-level program repair?, 2024.

[18] Junjielong Xu, Ying Fu, Shin Hwei Tan, and Pinjia He. Aligning the Objective of LLM-Based Program Repair . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2548–2560, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[19] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[20] Zheng Zhang, Yu Hao, Weiteng Chen, Xiaochen Zou, Xingyu Li, Haonan Li, Yizhuo Zhai, and Billy Lau. SymBisect: Accurate bisection for Fuzzer-Exposed vulnerabilities. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 2493–2510, Philadelphia, PA, August 2024. USENIX Association.

[21] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. {SyzScope}: Revealing {High-Risk} security impacts of {Fuzzer-Exposed} bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, 2022.

5 Appendix

Table 2: Patch Correctness

Setup	LLM	Plausible	Helpful	Wrong
SimpleAgent	GPT-4o	8	5	13
Function-Exploration	GPT-4o	2	2	1

5.1 Patch correctness

We manually verify the plausible correctness or helpfulness 31 random patches produced by our APR using GPT-4o, as shown in Table 2. As we performed manual verification, we could not determine if a patch is fully correct. We consider a patch plausibly correct if it follows the same semantics as the ground truth patch and prevents a crash, helpful if it does not properly address the root cause but targets the correct functions and prevents a crash, and wrong if it only prevents a crash but shares little to no similarity. We find that of the 31, 10 are plausibly correct, 7 are helpful, and 14 are wrong. This indicates that it is insufficient to simply rely on observing the absence of crashes to verify the correctness of patches. Interestingly, this result is consistent with what CrashFixer reported. Our rates of plausibly correct, helpful and wrong patches are 32.23%, 22.58%, and 45.16%, respectively, whereas the rates for CrashFixer are 32.91%, 15.18%, and 51.89%, respectively. This small study further suggests our simpler design achieved comparable performance to the much more complex solution.

5.2 Compute used for experiments

We use two machines for all tests. They are identical 56 core @ 2.3GHz, 160GB RAM, 1TB SSD. We run tests sequentially, such that a build uses all 56 cores, then 26 reproducer VMs use 52 cores and 52GB of RAM (2 cores, 2GB RAM each). The APR is very IO bound (to LLM APIs) and can be run on nearly anything. When reproducing kGym, it took 4 hours using a RTX 3060 and 400GB of space to generate BM25 indices. Table 3 shows compute times. Lower testing time for kGym tests can be attributed build failures ending the test early. Long test times for GPT-5 Thinking and Claude Opus 4.1 are likely due to their APIs being overloaded and forcing request retries as they had recently released, unfortunately we do not have a way of cutting that time out. They also take time to think and respond slower than GPT-4o. Preliminary testing and testing during development was also done on the same machines. We did not record time.

Table 3: Compute

Setup	LLM	Clock Hours
kgym-bm25	GPT-4-turbo	11.89
kgym-oracle	GPT-4-turbo	13.55
kgym-bm25	GPT-4o	13.71
kgym-oracle	GPT-4o	16.14
kgym-oracle+functionwise	GPT-4o	26.07
SimpleAgent-nobic	GPT-4o	45.59
SimpleAgent	GPT-4o	46.47
SimpleAgent+Feedback	GPT-4o	113.79
Function-Exploration	GPT-4o	45.99
SimpleAgent	Claude Opus 4.1	154.60
SimpleAgent	GPT-5 Thinking	121.24

5.3 Cost of KGym and GCP

KGym requires at least three GCP instances (scheduler, builder, reproducer), in varying shapes (2x c2-standard-16, 1x c2-standard-30) at a minimum hourly cost of $3.23 [4]. Running the minimum amount of GCP instances allows only one build job and one reproducer job to be run simultaneously, with a biweekly cost of at least $1087.47. This cost is unsustainable for many researchers (such as ourselves) and for intensive testing that may last multiple weeks, the money is much better spent on hardware.

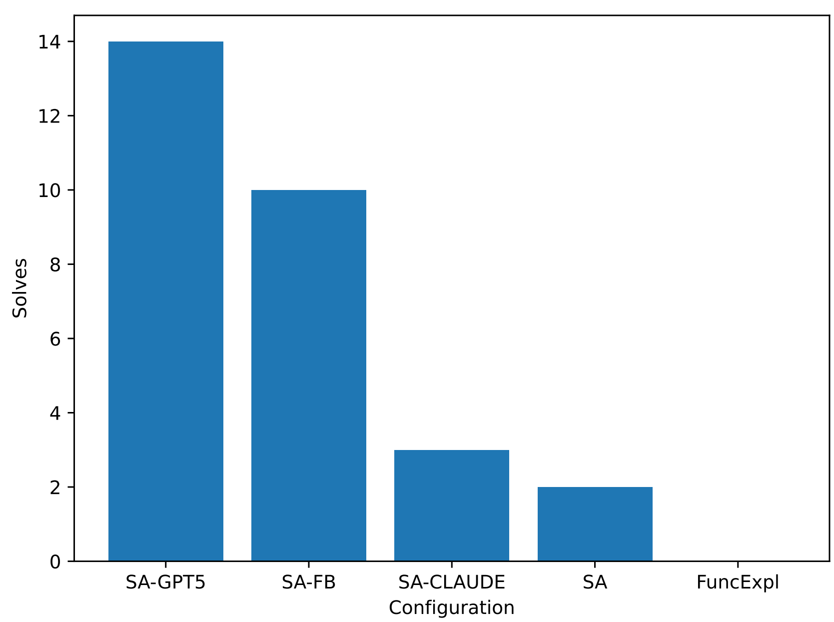

Figure 2: Unique solves per APR configuration

5.4 More evaluation

Unique Solves: Unique solves is an interesting metric that may be helpful to show versatility. In Figure 2 we see the most unique solves is achieved by SimpleAgent using GPT-5, which demonstrates the unique repair capability of the model not captured by other setups using other models. SimpleAgent using GPT-4o with feedback-driven retries also proves to be capable, solving 10 bugs neither GPT-5 nor Claude Opus 4.1 solved. SimpleAgent with Claude Opus 4.1 solves only 3 unique bugs, similar to our SimpleAgent using GPT-4o, although Claude performed much better overall. Our Function-

Table 4: Reproducer Job Output

Setup	LLM	Pass	Trigger	Racey	Boot Fail	Other
kgym-bm25	GPT-4-turbo	0	41	22	0	0
kgym-oracle	GPT-4-turbo	2	36	18	1	0
kgym-bm25	GPT-4o	2	58	35	0	0
kgym-oracle	GPT-4o	4	44	27	0	1
kgym-oracle+functionwise	GPT-4o	15	61	32	1	1
SimpleAgent-nobic	GPT-4o	25	85	57	5	0
SimpleAgent	GPT-4o	31	77	60	4	0
SimpleAgent+Feedback	GPT-4o	54	78	64	9	0
Function-Exploration	GPT-4o	22	87	65	2	0
SimpleAgent	Claude Opus 4.1	46	73	64	1	0
SimpleAgent	GPT-5 Thinking	62	60	60	0	0

Table 5: Build Job Output

Setup	LLM	Compilation Fails	Bad Patch
kgym-bm25	GPT-4-turbo	7	92
kgym-oracle	GPT-4-turbo	4	63
kgym-bm25	GPT-4o	2	78
kgym-oracle	GPT-4o	2	55
kgym-oracle+functionwise	GPT-4o	16	13
SimpleAgent-nobic	GPT-4o	26	2
SimpleAgent	GPT-4o	24	7
SimpleAgent+Feedback	GPT-4o	41	7
Function-Exploration	GPT-4o	24	8
SimpleAgent	Claude Opus 4.1	15	8
SimpleAgent	GPT-5 Thinking	15	6

Exploration Agent collected no unique solves, although this is expected due to its low pass rate, SimpleAgents specialization, and GPT-5's performance.

Compilation Failures: In Table 5 compilation failures remain consistent for our agents using GPT-4o, but we see a sharp drop when using SOTA LLMs. Even Claude Opus 4.1 substantially reduces compilation failures to match GPT-5 despite not meeting the same pass rate. The reduction in compilation errors indicates both LLMs have improved capabilities to maintain internal syntactic/semantic invariants when compared to GPT-4o, even if they do not match in other aspects such as reasoning. This suggests compilation failures can be used as a proxy metric for model reliability, or at least code generation consistency.

5.5 Background

5.5.1 Syzkaller

Syzkaller is an open-source coverage-guided kernel fuzzer developed by Google. It is designed to automatically discover security vulnerabilities, crashes, and unexpected behaviors in operating system kernels, with a primary focus on the Linux kernel, but it has also been adapted to other kernels like FreeBSD, NetBSD, Fuchsia, Darwin, and Windows. When a bug is found, Syzkaller is capable of outputting a reproducer program as a syz program and converting that syz program to a C program. These reproducers ideally can trigger the bug, although the reliability of the reproducer tends to vary, especially in the case of race conditions. Syzkaller has led to the discovery and reporting of thousands of Linux kernel bugs on a platform called Syzbot.

5.5.2 Syzbot

Syzbot is an automated bug reporting system built on top of Syzkaller and is also built by Google. Syzbot takes care of automatically triaging, reporting, and tracking bugs. It was created to reduce the manual effort needed in handling the large volume of crashes Syzkaller uncovers. Each bug entry in Syzbot has a unique ID, life cycle status (open, fixed, invalid), reproducers produced (if any), config for building, git commit, and sanitizer reports for each crash that occurs. Additionally, when the bug is fixed, the bug entry also contains the patch commit and occasionally the blamed bug inducing commit. Syzbot contains over 6500 fixed bugs and over 1500 open bugs for just the Linux kernel. This makes Syzbot an ideal source of bugs to create a benchmark.

5.5.3 kGym

kGym is similar RGym. The project introduces a gym, a dataset, and a basic APR. kGym itself is a kernel gym for automatically testing patches. It can orchestrate compiling kernels, applying patches, and running reproducers. kGym is highly dependent on GCP (Google Cloud Platform) as tests are run on GCP virtual machines. kGym's reliance on GCP makes it easily scalable, but impossible to run locally where compute is magnitudes cheaper. kGym's baseline APR operates in two modes. Assisted (or oracle) which uses the files from the accepted patch and unassisted which uses BM25 to retrieve files relevant to the bug. Unassisted and assisted modes achieve 0.72% and 5.38% pass rates on their benchmark dataset respectively.