KBOOTREPAIR: AUTOMATED LLM GUIDED REPAIR OF LINUX KERNEL BOOT FAILURES

by

RYAN GARFINKEL

A thesis submitted in partial fulfilment of the requirements

for the Honors Undergraduate Thesis program

in the College of Engineering and Computer Science

and in Burnett Honors College

at the University of Central Florida

Orlando, Florida

Spring 2026

Thesis Chair: Paul Gazzillo

© 2026 Ryan Garfinkel

ABSTRACT

The Linux kernel is an actively developed, highly configurable system software that supports a diverse range of platforms. Kernel fuzzers, like Google's syzkaller, have been embraced to find bugs, but they require bootable configurations that work in their environment. However, these tools rely on a set of configurations that exclude most code changes. Each option controls which code is included when the kernel compiles. Making modifications requires extensive knowledge, as there are over 15,000 interrelated configuration options. While prior work has used formal methods to guarantee configuration validity, this doesn't guarantee bootability. In this paper, we introduce KBootRepair, an iterative agent-based approach that repairs Linux kernel boot misconfigurations. It combines LLM boot suggestions with klocalizer, a formal reasoning tool, to generate bootable configurations. We evaluated KBootRepair on 50 random x86_64 configurations and achieved a 90% success rate within 20 attempts.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisors Dr. Paul Gazzillo and Dr. Aashish Yadavally for their guidance and encouragement throughout this project. Our frequent meetings helped me stay on track and develop a deeper understanding of the research process. Having the opportunity to present and discuss this work was an invaluable experience that helped me grow as a communicator. What started out as a chance for me to work at the intersection between LLMs and systems software ended up shaping my research trajectory and ambitions to pursue graduate school. I would also like to thank Sanan Hasonov for his mentorship on this project, guiding me from the initial planning stages to evaluating KBootRepair. Thank you for welcoming me to the lab and for always being available to talk through the challenges I was facing.

I would also like to thank my family for their endless support and inspiration throughout my life. My mother and father have worked hard to provide me with a good education and raise me to be well-rounded. To my grandmother, who shared her passion for biology and science with me, our talks about technology made me think deeper about the impact of what I do. To my friends at home, Ramsey Alsheikh, Justin Berke, and Ritvik Teegavarapu, you have inspired me to work harder and step outside my comfort zone every day. This thesis, along with other personal accomplishments, would not have been possible without you.

TABLE OF CONTENTS

LIST OF FIGURES....vii

LIST OF TABLES viii

INTRODUCTION 1

BACKGROUND....4

Misconfiguration Classification 4

Configuration Validity.... 4

Coverage Guided Testing.... 5

LLMs For Configurable Software Tasks.... 5

NAIVE LLM CASE STUDY 6

Setup 6

Results.... 7

KBOOTREPAIR DESIGN 8

Overview.... 8

Feedback Loop.... 9

Tooling 9

Agent Output 10

EVALUATION....11

Experimental Setup 11

Success 11

Effectiveness.... 12

Configuration Changes 14

LIMITATIONS 16

RELATED WORK 17

Crash Resolution 17

Kernel Testing 17

CONCLUSION 18

REFERENCES 19

LIST OF FIGURES

Figure 1 - Naive LLM Approach 6

Figure 2 - KBootRepair Architecture.. 8

Figure 3 - Attempt Distribution.. 12

Figure 4 - Total Constraints vs Edit Distance 14

LIST OF TABLES

Table 1 - Naive LLM Approach Results 7

Table 2 - Agent Tools 9

Table 3 - KBootRepair Success Breakdown 11

Table 4 - Attempt Distribution Summary Statistics 12

Table 5 - Overall Token Breakdown of All Samples 13

Table 6 - Attempt Token Breakdown of All Samples 13

Table 7 - Edit Distance Summary Statistics 14

Table 8 - Constraints Breakdown Summary Statistics 14

INTRODUCTION

Operating systems are the bedrock of today's computer infrastructure. They are designed to be configurable and to be used in a diverse range of hardware platforms and environments. The Linux kernel, known for its extreme configurability, has over 15,000 options that support all aspects of the system such as memory management and scheduling [1]. Each of these controls which code gets compiled at build time. Additionally, configuration options are interrelated. Changing one option's value can introduce conflicts or dependency errors with other options, opening the door to possible misconfigurations. This poses a challenge to those building the system, where slight changes can cause system failures and can be expensive to resolve [2].

The Linux kernel is actively developed, receiving thousands of commits per month [3]. Testing these changes is vital for identifying bugs and vulnerabilities that may be introduced. However, the configurability space is extremely large, even after considering the constraints between options. Run-time testing is especially difficult because it requires a bootable configuration tailored to the testing platform. This is important because kernel fuzzing is utilized to identify vulnerabilities in the kernel [1]. One of the tools that does this is syzkaller [4], Google's coverage guided fuzzer. It has found about 4,000 bugs and is also considered a top bug reporter [1]. However, syzkaller, along with other kernel fuzzers, relies on a small set of configurations that exclude most of the kernel code [1], limiting the bugs they can find and leaving parts of the kernel untested.

To reach different parts of the kernel, custom, bootable configurations are needed. While some prior work [3] has focused on navigating the relationships between options to generate valid configurations, it doesn't guarantee the kernel will boot on a given platform, as explored in

this work. Bootability is dependent on configuration validity, the platform it is on, and the run-time behavior of the code. This makes bootability repair its own problem that needs to be solved.

In recent years, Large Language Models (LLMs) have been studied in a variety of configurable software tasks including kernel tuning [5, 6], validation [7], and repair [8]. While they have shown promise, LLMs have their limitations. A naive approach that involves passing the entire configuration, build log, and boot log into an LLM, asking it to generate a new configuration file is unsuitable. Each file can be thousands of lines long, exploding the input and output token use. Additionally, the validity of the configuration depends on satisfying dependencies and conflicts between the options, which if not met could result in a configuration that doesn't build, let alone boot.

In this paper, we introduce KBootRepair, an agentic approach that combines LLMs and formal reasoning methods to repair boot misconfigurations. Our key insight is to repair these misconfigurations by breaking the solution into two stages: (1) identifying relevant options causing the boot failure using the LLM and (2) generating a valid configuration using formal reasoning methods guided by the LLM's suggestions. KBootRepair isolates the agent to identify boot suggestions, a subset of options to keep included or excluded. These options are then directly passed into the klocalizer [3] tool to generate a new configuration. KBootRepair then tries to build and boot the kernel to validate the changes. It repeats these steps until a bootable configuration is made, updating the agent's prompt each time to provide context of previous attempts.

We then evaluate KBootRepair on a set of unbootable configurations. We started by first generating 50 random configurations, none of which booted. We then passed these

configurations into the KBootRepair tool which successfully repaired 90% of the samples in an average of 10.76 iterations.

This paper makes the following contributions:

- The KBootRepair architecture, which combines LLM generated boot suggestions with formal reasoning methods to iteratively repair unbootable configurations.
- The implementation of the KBootRepair tool.
- An evaluation of the KBootRepair tool that shows 90% of the samples were repaired in 10.76 attempts on average, spending on average 2,340,126 tokens per repair.

Our artifact is publicly available on GitHub and contains the complete implementation, evaluation, and data collection scripts.

BACKGROUND

Misconfiguration Classification

Prior work has extensively studied misconfigurations in systems software. Yin et al. [2] created a taxonomy to classify types of configuration errors, examine their impact, and observe the system's response to them. They identified three kinds of configuration errors: compatibility, component, and parameter. Compatibility misconfigurations (14.5%-30%) typically occur when different components or modules don't work with each other, while component issues typically occur when modules are missing, files are misplaced or incorrectly formatted, or when there are insufficient resources [2]. Parameter issues, estimated to make up 70%-85.5% of configuration errors, either implicitly or explicitly violate semantic rules (illegal) or intended functionality (legal) [2]. This suggests that despite meeting formal semantic requirements, misconfigurations are still possible, which can have effects on system availability [2].

Configuration Validity

With over 15,000 interrelated options to choose from [1], Linux kernel validity can be a challenge. Fortunately, prior work [3, 9] has developed ways using SAT solvers to automatically satisfy constraints using Kconfig, the Linux kernel's configuration specification language [10]. Krepair [3], created by Yildiran et al., is a tool that updates configuration files to achieve a higher patch coverage, while changing less than 1.53% of the original options in 99% of patches. ConfigFix [9] is another tool that utilizes SAT solvers to resolve configuration conflicts.

Coverage Guided Testing

With thousands of commits each month [3], it is vital for the Linux kernel to have effective automatic testing to discover new bugs. Developers have embraced fuzzing methods which feed continuous random input into the kernel to discover vulnerabilities [1]. Google's syzkaller [4] tool, for example, does this and has become a top reporter of finding new bugs, about 4,000 [1]. But these kinds of tools require a bootable configuration to work that covers incoming changes. Unfortunately, prior work by Hasanov et al. noted configuration selection strategies, selecting Linux's defconfig, randconfig, allnoconfig, and even configurations provided by syzkaller, exclude most code changes [1, 3].

LLMs For Configurable Software Tasks

In recent years, large language models (LLMs) have been studied in a variety of configurable software tasks such as validation [7], performance tuning [5, 6], and repair [8]. While these works have shown LLMs have great promise and ability for performing these tasks, Spieker et al. found that off-the-shelf models have difficulty understanding small performance differences between configurations [11]. Lian et al. also noted that Ciri [7] struggles with option dependency violations when validating configuration. In contrast, works like AutoOS [5] and BYOS [6], which pair LLMs with knowledge graphs, have outperformed vanilla LLMs on performance tuning tasks.

NAIVE LLM CASE STUDY

Setup

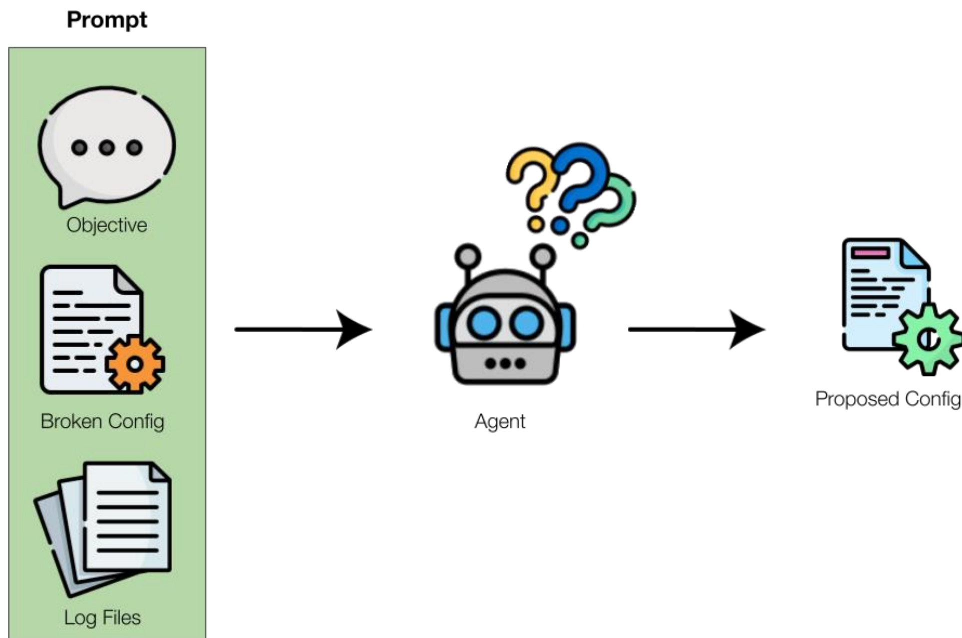

Figure 1 - Naive LLM Approach

As an initial approach, we evaluated a naive, one-shot LLM method intended to repair boot misconfigurations. We sampled 5 random, x86_64, unbootable configurations and captured their build and boot logs. Then, as shown in Figure 1, we directly passed the entire configuration and log file contents into an LLM and asked it to generate a new bootable one. We used the gemini-3.1-pro-preview model [12] to perform these repairs. Afterward, we took the LLM generated configuration file, ran the kernel's "make olddefconfig" to resolve invalid configuration options, then attempted to build and boot it. We used QEMU to boot the kernel, giving it at most 5 minutes before considering it a failure.

Results

Table 1 - Naive LLM Approach Results

Sample	Build Success	Boot Success	Input Tokens	Output Tokens	Total Tokens	Hit Token Limit
1	True	False	294,688	65,532	360,220	True
2	True	False	801,398	51,117	852,515	False
3	True	False	552,152	65,532	617,684	True
4	True	False	239,081	65,532	304,613	True
5	True	False	185,163	65,532	250,695	True

The results of this approach can be summarized in Table 1. We tracked whether the configuration built, booted, the input, output, and total token consumption, as well as whether the LLM stopped early. Ultimately, we found that this approach was unsuitable for two reasons: (1) it was unable to repair any of the samples and (2) it had high token consumption. While all 5 configurations built, none of them booted. 4 out of the 5 generated configurations had kernel panics in their boot log. The fifth sample did not log any kernel panics but also did not boot in the 5 minute period, suggesting it might have needed more time.

On average, each repair consumed 414,496 input tokens and 62,649 output tokens per repair, costing about $2.79 each. After looking at the configurations themselves, we noticed 4 out of the 5 were incomplete, some ending halfway through a line with "# CONFIG_". During these repairs, the LLM reached Google's output token limit [12] on the gemini-3.1-pro-preview model, leaving generated configurations incomplete, highlighting that the output is too large for the LLM's output context window.

KBOOTREPAIR DESIGN

Overview

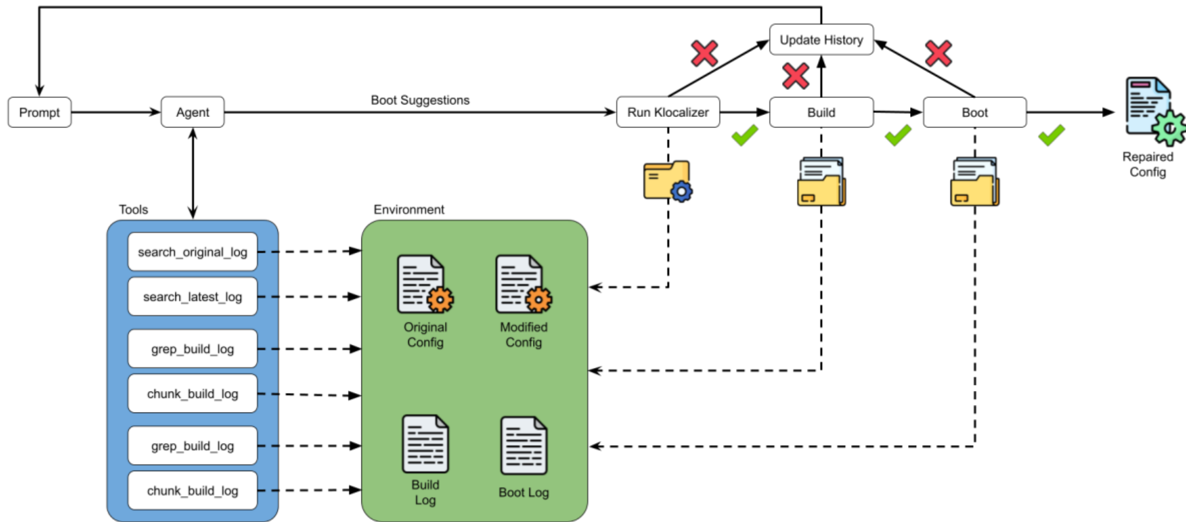

Figure 2 - KBootRepair Architecture

To overcome the failures of the naive LLM approach, KBootRepair takes an iterative, tool-augmented approach combining LLM boot suggestions with formal reasoning methods to generate repaired, bootable configurations. It addresses the limitations of the one-shot approach and high token consumption, shown in Figure 2. First, KBootRepair introduces an iterative loop, allowing the agent to refine the configuration over time. After each attempt, its history is updated, which is appended to the prompt. Second, the agent is given tools to query the configurations rather than passing the entire files into the prompt, saving input tokens. Third, the agent outputs boot suggestions, a subset of options to either include or exclude, rather than an entirely new configuration file. These options are directly passed into the klocalizer [3] tool as

constraints to produce a valid configuration. KBootRepair then verifies the configuration boots, updating the environment in case it needs to loop again.

Feedback Loop

When KBootRepair first starts, a verify step runs to ensure the configuration needs repair and to generate the initial build and boot logs, used as input into the first attempt. Each iteration starts by invoking the agent, which utilizes the tools available to produce boot suggestions. These suggestions are then passed into the klocalizer [3] tool to generate a new configuration. Finally, KBootRepair tries to build and boot the kernel, updating the build and boot logs for the next iteration if the kernel fails to boot.

Tooling

Table 2 - Agent Tools

Tool Name	Arguments	Description
search_original_config	options: list[str]	Returns option values of from original config.
search_latest_config	options: list[str]	Returns option values of from latest config.
grep_build_log	pattern: str	Returns build log lines matching the pattern.
chunk_build_log	line: int	Returns build log lines surrounding line number.
grep_boot_log	pattern: str	Returns boot log lines matching the pattern.
chunk_boot_log	line: int	Returns boot log lines surrounding line number.

Rather than passing the entire configuration and log files into the prompt, as done in the naive approach, we provided KBootRepair with tools to query the contents of these files, to

reduce token usage. We created six tools: two for querying the original and latest configuration files and four for retrieving content from the build and boot logs. As shown in Table 2, the configuration search tools accept a list of option names and return their values from the respective file. The grep log tools accept a regular expression and return the lines matching the pattern, with line numbers appended. The chunk log tools return the 50 lines surrounding the line of interest for more context. In some instances, when klocalizer fails to generate a new configuration or a build fails leaving no boot log, tools become conditionally available.

Agent Output

To address the output token limit challenge we faced in the naive approach, KBootRepair asks the agent to output suggestions rather than an entire configuration file. These suggestions contain a list of options to include and exclude, along with a reasoning string. The list of includes and excludes is directly passed into klocalizer [3], offloading the job of satisfying option constraints and generating a new, valid configuration to the tool. The reasoning string is preserved and passed into the prompt for the next iteration if the kernel fails to boot.

EVALUATION

Experimental Setup

We evaluated KBootRepair on a sample of 50 unbootable, randomly generated x86_64 configurations on the Linux kernel v7.0-rc1. Of the 50 unbootable samples, 25 were unable to build successfully. We used QEMU to boot each configuration, allowing it five minutes before considering the attempt a failure. To limit the token usage, we gave KBootRepair a maximum of 20 attempts to get the configuration bootable. To generate the suggestions, we used the gemini-3.1-pro-preview model [12] from Google.

Success

Table 3 - KBootRepair Success Breakdown

Outcome	Count
Normal Boot	45
Maintenance Boot	1
Failed within 20 attempts	4
Total	50

As shown in Table 3, KBootRepair successfully repaired 45 out of the 50 samples, achieving a 90% success rate. One configuration booted into maintenance mode, enough to log the user in to resolve the boot failures manually, and the remaining four failed to boot within the 20 attempt limit. These results demonstrate KBootRepair’s ability to repair configurations that could not initially build and configurations that could build but not boot.

Effectiveness

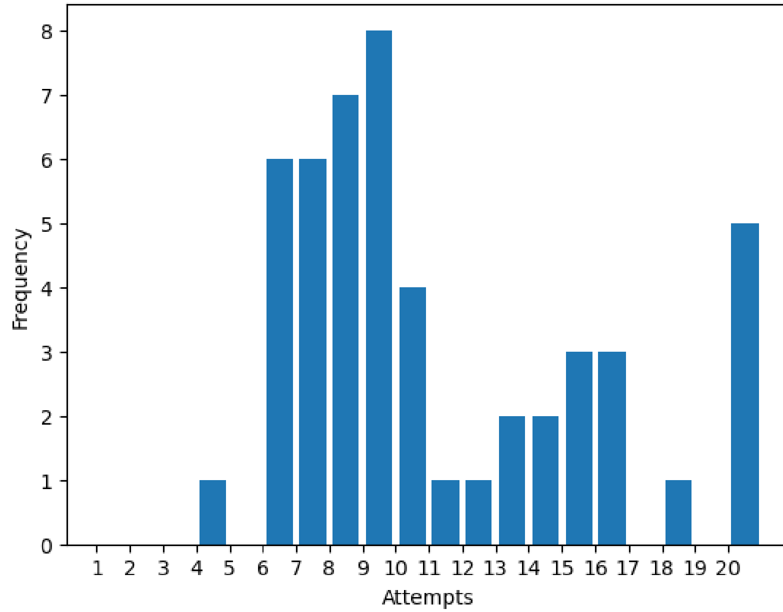

Figure 3 - Attempt Distribution

Table 4 - Attempt Distribution Summary Statistics

Min	Median	Max	Mean	Standard Deviation
4	9	20	10.76	4.51

As shown in Figure 3 and Table 4, the number of attempts KBootRepair needed to repair the configurations centered around 9, with an average of 10.76, with some requiring more. Notably, the minimum number of attempts was 4, suggesting bootability repair cannot be done in a single iteration. Instead, it requires testing and refining changes. The five samples that reached 20 attempts correspond to the one sample that booted in maintenance mode and the other four that failed to boot.

Table 5 - Overall Token Breakdown of All Samples

	Min	Median	Max	Mean	Standard Deviation
Input Tokens	534,849	1,587,745	6,366,997	2,241,869	1,582,162
Output Tokens	25,505	69,557	288,135	98,256	69,861
Total Tokens	577,852	1,678,990	6,625,692	2,340,126	1,650,121

Table 6 - Attempt Token Breakdown of All Samples

	Min	Median	Max	Mean	Standard Deviation
Input Tokens	7,608	187,102	869,662	208,352	121,101
Output Tokens	473	7,608	36,683	9,131	5,768
Total Tokens	8,114	193,729	874,324	217,483	124,915

As shown in Tables 5 and 6, token usage varied across attempts and repairs. These numbers also reflect all 50 samples, including the five that failed to boot normally within 20 attempts. On average, KBootRepair consumed 2.3 million total tokens. Of the 45 that booted normally, the average was 1.9 million total tokens. It is also important to note the per attempt token consumption, since each attempt invokes the agent once like in the naive approach. On average, each attempt on all 50 samples consumed 208,352 input tokens and 9,131 output tokens. Compared to the naive approach (Table 1), this is a significant drop from 414,496 input tokens (~50%) and 62,649 output tokens (~85%) on average each time the LLM is invoked.

Configuration Changes

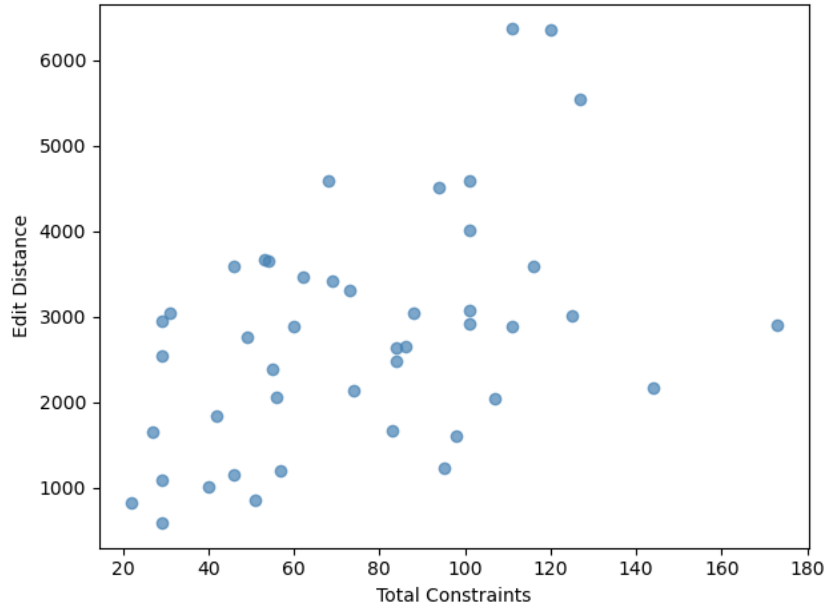

Figure 4 - Total Constraints vs Edit Distance

Table 7 - Edit Distance Summary Statistics

Min	Median	Max	Mean	Standard Deviation
584	2878	6364	2797.7	1361.9

Table 8 - Constraints Breakdown Summary Statistics

	Min	Median	Max	Mean	Standard Deviation
Includes	14	42	88	42.6	21
Excludes	6	29	108	33	23.9
Total	22	73	173	75.6	35.2

We measured the number of constraints and edit distance, summarized in Figure 4 and Tables 7 and 8, to understand the modifications KBootRepair made to the 45 successfully repaired samples. The constraints are the total number of options suggested by the agent to

include and exclude, and the edit distance is the number of options that changed between the original and repaired configurations measured with the kernel's diffconfig tool. On average, the agent suggested changing 75 options to include or exclude from the configuration, which ultimately edited 2797 options on average when klocalizer [3] made changes. When looking at Figure 4, we noticed there appeared to be a weak, positive correlation between constraints and edit distance. However, there appears to be high variance, suggesting the relationship is determined by how klocalizer [3] selects satisfiable configurations.

LIMITATIONS

We limited our evaluation of KBootRepair to samples from the x86_64 architecture, on version 7.0-rc1 of the Linux kernel and do not know how repairs may generalize to other architectures and versions. Furthermore, we only used the gemini-3.1-pro-preview model, whose knowledge cutoff is January 2025 [12], to generate boot suggestions. Performance may differ between similar size or smaller models like Gemini Flash. It is also important to note that token usage can vary each time the agent is invoked. The number of tools that are invoked, the amount of content returned from the tool calls, and the number of attempts can increase token consumption.

We gave QEMU five minutes to boot the kernel when determining if the configuration was successfully repaired. Configurations that required longer to boot would have been marked as a failure, increasing the number of attempts and token usage. Additionally, we believe that the reasoning string appended to the prompt after each iteration, if incorrect, could negatively mislead future attempts. The quality of the build and boot logs can also affect the ability of the agent to reason about the real boot issue. Empty logs could result in less informed boot suggestions.

RELATED WORK

Crash Resolution

Beyond code generation, LLMs have been used for repairing Linux kernel bugs found during fuzzing. CrashFixer [8], made by Mathai et al., does this. It is an iterative tool that hypothesizes and applies patches, resolving ~49% of crashes with file localization assistance [8]. It works by passing in the crash report and the source code of likely files causing the crash. CrashFixer then iteratively hypothesizes, generates patches, selects the best compilable candidate, and tests until the crash is resolved, using the latest crash report to improve its output between attempts [8]. KBootRepair takes a similar approach, but targets configuration repair instead of source code repair. Prior work has also focused on evaluating LLMs for Linux kernel crash resolution. LIVE-KBENCH [13], created by Huang et al., is an evolving benchmark that evaluates agents performing crash resolution repairs with new bugs found by syzbot.

Kernel Testing

KernelGPT [14], developed by Yang et al., is a tool that generates and repairs Syzkaller specification files. Its goal is to enhance fuzzing by iteratively refining specifications from validation feedback. With this, they were able to find 24 unknown bugs [14]. Like KBootRepair, their approach is iterative, refining output based on feedback from syz-extract and syz-generate tools [14]. However, their goal was to specifically improve kernel fuzzing, while KBootRepair targets kernel configuration repair.

CONCLUSION

In summary, this study introduces KBootRepair, an iterative, agent-based tool that combines LLM boot suggestions with formal reasoning tools to repair unbootable configurations. We evaluated KBootRepair on a sample of 50 random, unbootable x86_64 Linux kernel configurations and achieved a 90% success rate, taking an average of 10.76 attempts per repair. As for next steps, we intend to test KBootRepair on the arm64 architecture and with different models to understand how performance is affected. We would also like to add additional tool parameters that allow the user to define hard constraints, options that must be included or excluded, and an option to pass in a patch file the agent can query to gear suggestions towards these code changes. KBootRepair demonstrates that LLMs paired with formal reasoning tools have the ability to iteratively repair boot misconfigurations.

REFERENCES

[1] Sanan Hasanov, Stefan Hagy, and Paul Gazzillo. 2025. A Little Goes a Long Way: Tuning Configuration Selection for Continuous Kernel Fuzzing. In 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE Computer Society, Los Alamitos, CA, USA, 795-807. DOI:https://doi.org/10.1109/ICSE55347.2025.00042

[2] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP ’11), Association for Computer Machinery, Cascais, Portugal, 159-172. DOI:https://doi.org/10.1145/2043556.2043572

[3] Necip Fazil Yildiran, Jeho Oh, Julia Lawall, and Paul Gazzillo. 2024. Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times. Proc. ACM Softw. Eng. 1, FSE (July 2024). DOI:https://doi.org/10.1145/3643746

[4] Google and syzkaller contributors. 2025. syzkaller. Retrieved from: https://github.com/google/syzkaller

[5] Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, Qi Guo, and Yunji Chen. 2024. AutoOS: make your OS more powerful by exploiting large language models. In Proceedings of the 41st International Conference on Machine Learning (ICML ’24), JMLR.org, Vienna, Austria.

[6] Hongyu Lin, Yuchen Li, Haoran Luo, Kaichun Yao, Libo Zhang, Zhenghong Lin, Mingjie Xing, Yanjun Wu, and Carl Yang. 2026. BYOS: Knowledge-driven Large Language Models Bring Your Own Operating System More Excellent. Retrieved from https://arxiv.org/abs/2503.09663

[7] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. 2025. Large Language Models as Configuration Validators. In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25), IEEE Press, Ottawa, Ontario, Canada, 1704-1716. DOI:https://doi.org/10.1109/ICSE55347.2025.00017

[8] Alex Mathai, Chenxi Huang, Suwei Ma, Jihwan Kim, Hailie Machell, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. 2025. CrashFixer: A crash resolution agent for the Linux kernel. Retrieved from https://arxiv.org/abs/2504.20412

[9] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgen Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 91-100. DOI:https://doi.org/10.1109/ICSE-SEIP52600.2021.00018

[10] Paul Gazzillo. 2017. Kmax: finding all the configurations of Kbuild makefiles statically. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), Association for Computing Machinery, Paderborn, Germany, 279-290. DOI:https://doi.org/10.1145/3106237.3106283

[11] Hekge Spieker, Théo Matricon, Nassim Belmecheri, Jørn Eirik Betten, Gauthier Le Bartz Lyan, Heraldo Borges, Qeuntin Mazouni, Dennis Gross, Arnaud Gatlieb, and Mathieu Acher. 2025. Prompting for Performance: Exploring LLMs for Configuring Software. Retrieved from https://arxiv.org/abs/2507.09790

[12] Google. 2026. Gemini 3.1 Pro Preview. Retrieved from https://ai.google.dev/gemini-api/docs/models/gemini-3.1-pro-preview

[13] Chenxi Huang, Alex Mathai, Feiyang Yu, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Eugene Wu, Kostis Kaffes, Junfeng Yang, and Baishakhi Ray. 2026. Outrunning LLM Cutoffs: A Live Kernel Crash Resolution Benchmark for All. Retrieved from https://arxiv.org/abs/2602.02690

[14] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2025. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS ’25), Association for Computing Machinery, Rotterdam, Netherlands, 560-573. DOI:https://doi.org/10.1145/3676641.3716022