

KBootRepair: Automatically Repairing Linux Kernel Boot Misconfigurations

Ryan Garfinkel¹ Paul Gazzillo² Sanan Hasonov³

¹University of Central Florida, Orlando, FL

Motivation

- The Linux kernel is widely used because of configurability.
 - IoT devices, web servers, and supercomputers.
- 20,000+ interrelated options.
- Options control which code gets included in the kernel.
- Actively developed, hundreds of new commits per month.
- Kernel needs to be tested constantly to find new bugs.
- Default configurations exclude most changes.
- Boot misconfigurations** occur when kernel settings are not suited to testing platform.

Background

- KConfig specifies relationships between kernel options.
- Valid options $\neq$ kernel build $\neq$ kernel boot
- Krepair changes configurations, following KConfig rules.
 - Can also specify options to include/exclude.
- ConfigFix resolves dependency issues.
- LLMs have been used for kernel tuning and validation.

Naive Approach

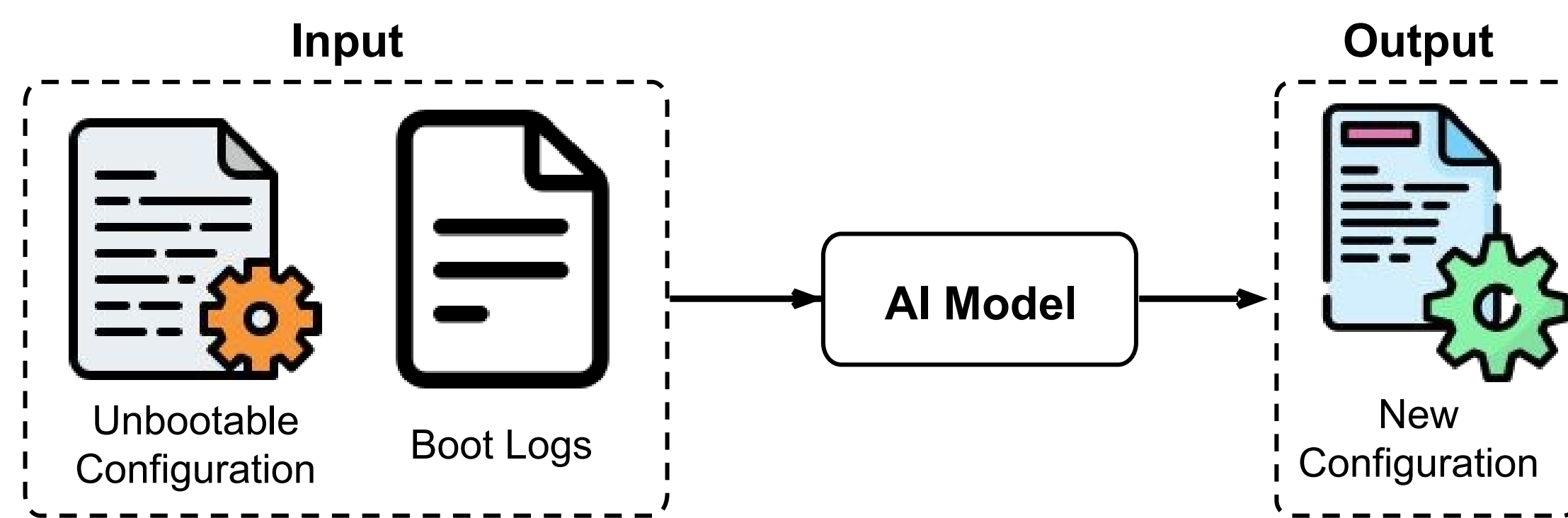


Figure 1. Naive approach steps

- Tested on 5 random x86 configurations that didn't boot.
- Prompted Gemini 3.1 Pro model to generate new configuration.
- Spent an average of 461,789 input tokens each time.
- Reached Gemini Pro's maximum output tokens (65,532) each time.

KBootRepair Architecture

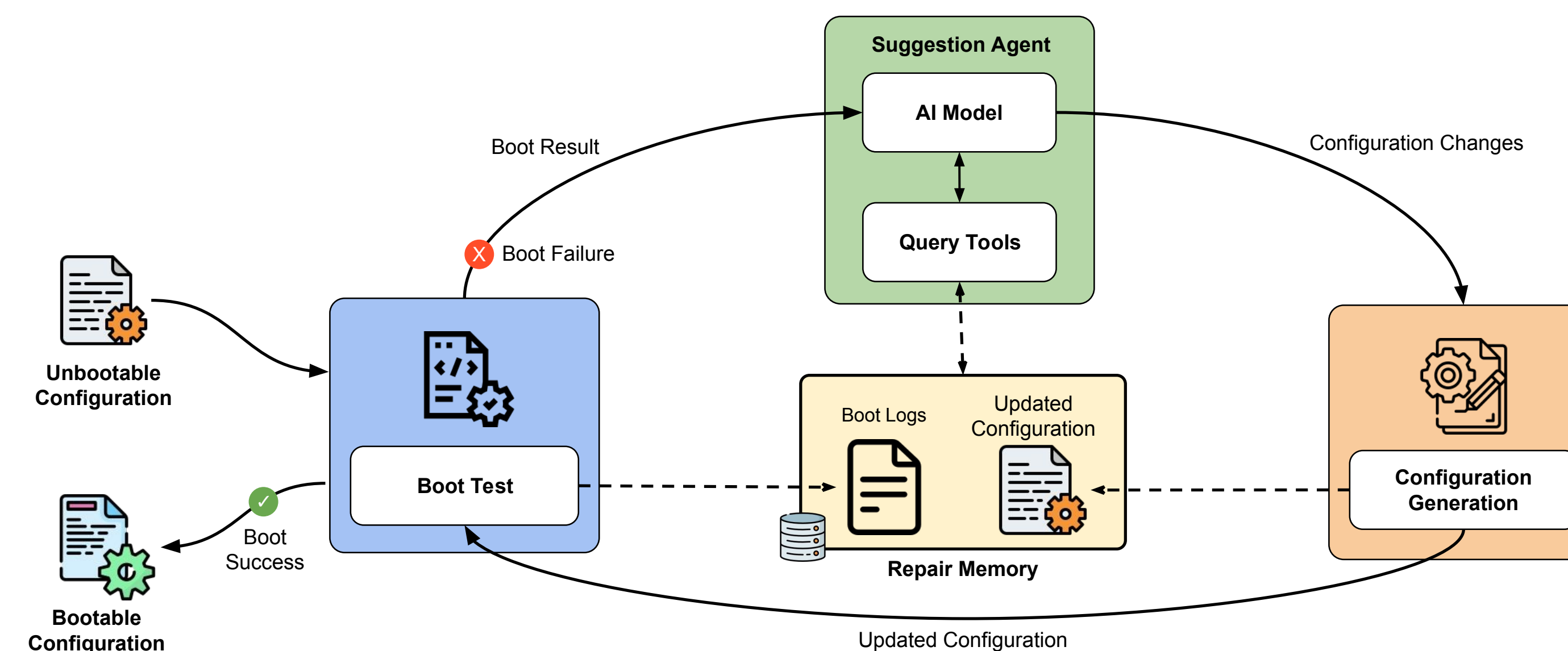


Figure 2. KBootRepair architecture design.

Tool Name	Description
search_original_config	Get values from the original configuration.
search_latest_config	Get values from the latest configuration.
grep_build_log	Get build log lines matching a pattern.
chunk_build_log	Get build build log lines near a given line.
grep_boot_log	Get boot log lines matching a pattern.
chunk_boot_log	Get boot boot log lines near a given line.

Table 1. Suggestion Agent query tool list.

Field	Contents
Verifier Result	success, no-satisfying-constraints, error, or not-run
Build Result	success or failed
Boot Result	yes, no, maintenance, panic, or timeout
Defined	The list of options that were enabled.
Undefined	The list of options that were disabled.
Reasoning string	Agent-reported explanation for the suggestions.

Table 2. Information obtained after each attempt.

Experimental Setup

- 100 random kernel configurations (50 x86, 50 ARM).
 - 27 configurations didn't initially build.
- 5 minutes to boot before considering it a boot failure.
- Maximum 20 tools calls per attempt.
- Maximum 20 iterations per repair.
- Used Gemini 3.1 Pro model for repair.
- Used Gemini 3.5 Flash model to compare against Pro.

Evaluation

- 92% success rate (90% x86, 94% ARM).
- Median 9 attempts for all repairs.
- Mean 1.85 million tokens (\$4.41) per repair.
 - 60% less input tokens per attempt than naive approach.
 - 89% less output tokens per attempt than naive approach.
- Repair time tasks 5.10 hours on average.
 - Most time (68.3%) spent on building the kernel.
- Mean 85% options preserved in repaired samples.
- Sample variety increased slightly in repaired samples.
 - 6% for x86 samples, 4% for ARM samples.
- Performance decreased when switching to Flash model.
 - Success rate dropped to 78%.
 - Token consumption increased with more attempts.
 - Slightly increased preservation with Gemini Flash.
 - Similar sample variety.

Conclusion

- Automates repair of Linux kernel misconfigurations.
- Combines LLM suggestions with formal reasoning.
- Preserves majority of original configuration options.
- Can be used to improve support testing.

Results

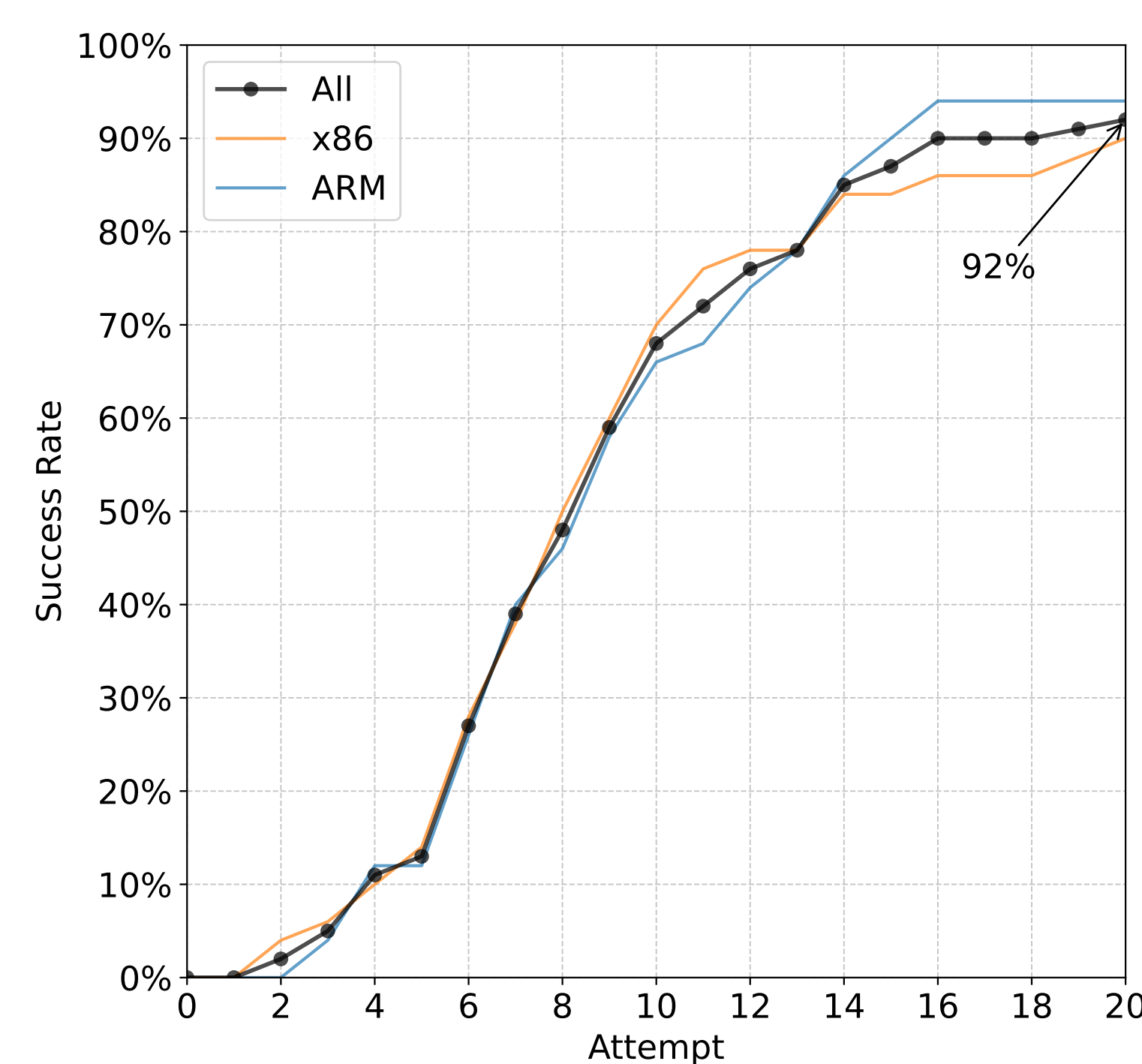


Figure 3. Boot success versus attempts.

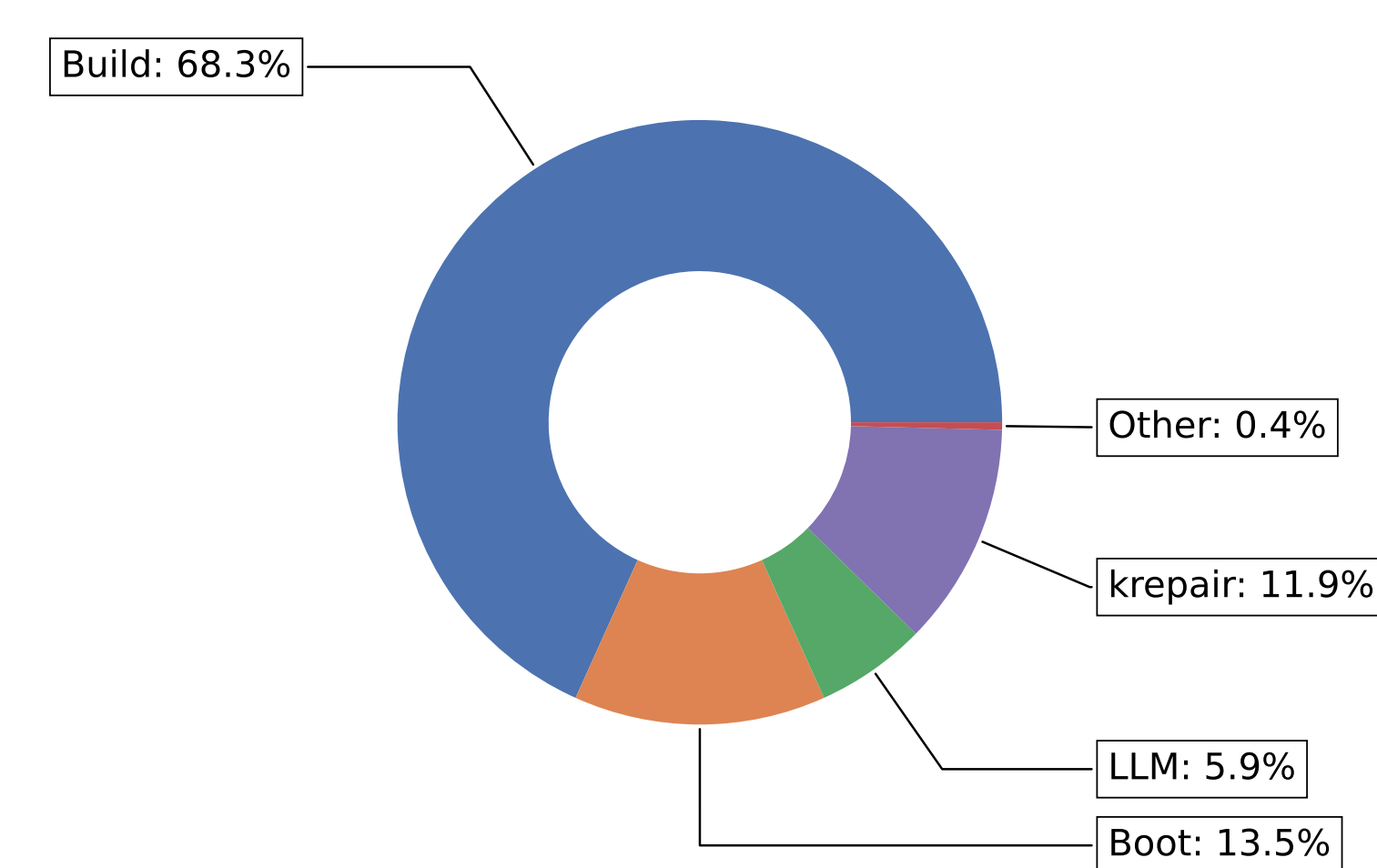


Figure 4. Overall distribution for each repair step.

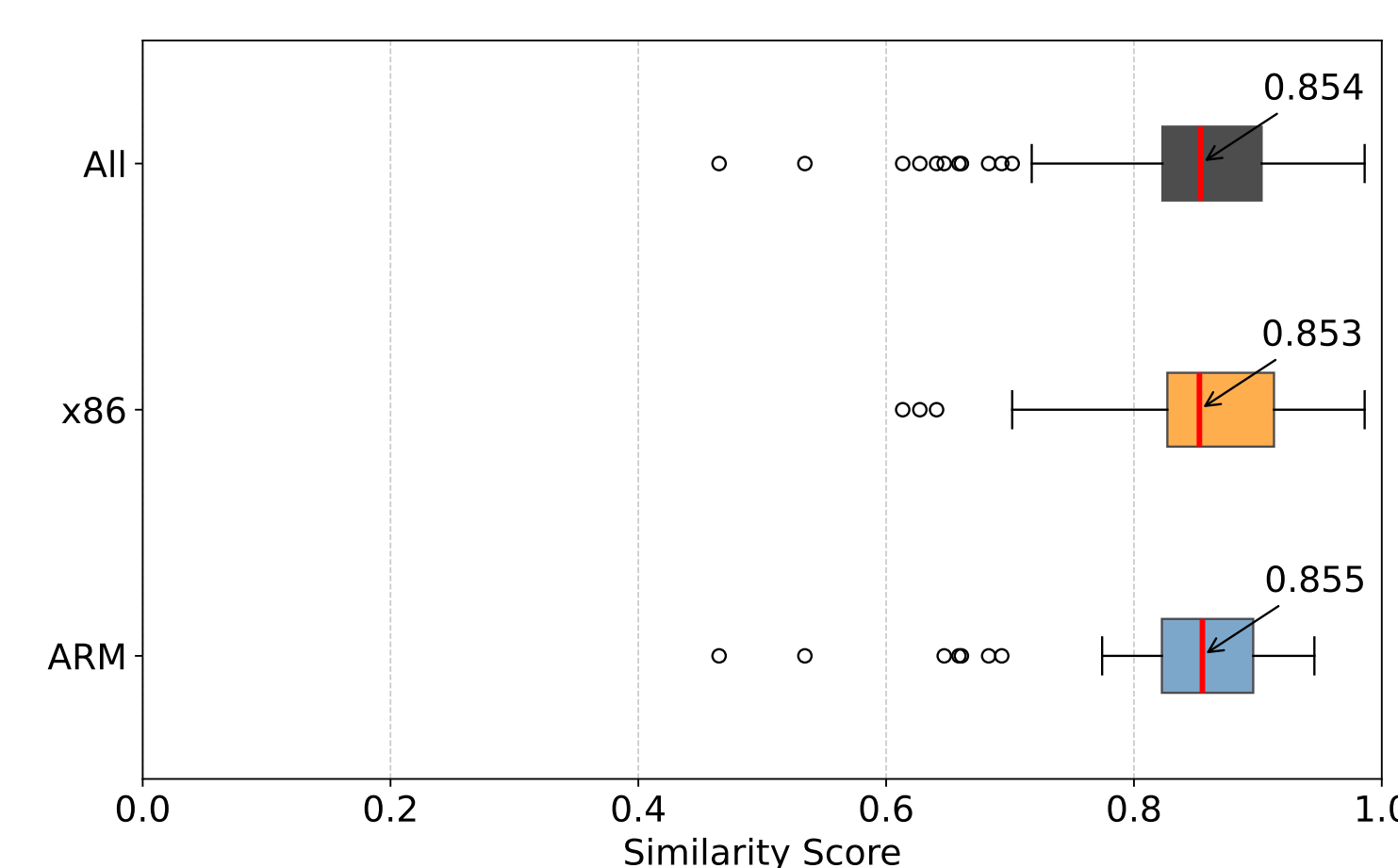


Figure 5. Similarity distribution between repaired samples before/after.

Metric	gemini-3.1-pro	gemini-3.5-flash
Success Rate	90% (45/50)	78% (39/50)
Attempts	9.74 (5.06)	12.0 (5.72)
Input Tokens (per repair)	1.93 mil (1.68 mil)	2.82 mil (2.12 mil)
Output Tokens (per repair)	82.0k (74.4k)	104.7k (82.8k)
Total Tokens (per repair)	2.01 mil (1.75 mil)	2.92 mil (2.20 mil)
Duration (hr)	4.37 (2.83)	4.48 (2.66)
Preservation	0.85 (0.09)	0.87 (0.09)
Sample Similarity (after)	0.33 (0.04)	0.33 (0.04)

Table 3. x86 Gemini Pro and Gemini Flash comparison.