Barredo, J., Eceiza, M., Flores, J.L., Iturbe, M. (2026). Sow Smarter, Not Harder: Evaluating LLM-Generated Seeds for Fuzzing Critical Infrastructure. In: Bergström, E., Hämmerli, B., Kitkowska, A., Kävrestad, J. (eds) Critical Information Infrastructures Security. CRITIS 2025. Lecture Notes in Computer Science, vol 16291. Springer, Cham. https://doi.org/10.1007/978-3-032-19540-1_17

Sow Smarter, Not Harder: Evaluating LLM-generated Seeds for Fuzzing Critical Infrastructure

Jorge Barredo[1,2], Maialen Eceiza[1], Jose Luis Flores[3], and Mikel Iturbe[2]

[1] IKERLAN Technology Research Centre, 20500 Arrasate/Mondragón, Spain
[2] Mondragon Unibertsitatea, 20500 Arrasate/Mondragón, Spain
[3] University of the Basque Country, 20018 Donostia/San Sebastián, Spain

Abstract. Software vulnerabilities in critical infrastructure components can lead to severe disruptions. While fuzzing effectively identifies such weaknesses, the quality of initial seed inputs significantly impacts its effectiveness. This study evaluates how large language models (LLMs) can generate better fuzzing seeds for critical infrastructure software. We compared seven LLMs—ChatGPT-4-Turbo, Claude 3.0 Opus, Claude 3.7 Sonnet, DeepSeek-V3, Gemini 2.0 Flash, Grok 3, and Mistral 7B—with manual baselines across six programs, including industrial control libraries, routing components, and network firmware. Over 20 independent 24-hour campaigns per model and program, LLM-generated seeds achieved 14.8% higher code coverage, detected 56.3% more unique crashes, and reached first crashes 373.9% faster than manual methods. Performance patterns emerged across different infrastructure protocols, with certain models excelling at complex SCADA data formats while others performed better for network security components. The 56.5% computational efficiency improvement benefits resource-constrained operational technology environments. These findings demonstrate that LLM-generated seeds can meaningfully enhance vulnerability detection in software underlying critical infrastructure systems, offering a practical approach to strengthening resilience against cyber threats.

Keywords: LLMs · Fuzzing · Vulnerability detection.

1 Introduction

Critical infrastructure (CI) systems, such as power grids and healthcare facilities, depend on software, particularly embedded systems, to operate reliably [19]. Moreover, secure communication libraries like OpenSSL are essential for network servers supporting CI operations. However, software vulnerabilities can cause severe disruptions, as seen in the 2021 Colonial Pipeline attack, which interrupted fuel distribution, and the 2017 WannaCry ransomware attack, which affected 200,000 healthcare systems across 150 countries [18]. Thus, detecting vulnerabilities during software development is essential to protect CI systems.

Fuzz testing, or fuzzing, stands out as an effective technique for identifying software vulnerabilities. By systematically providing programs with malformed

inputs, fuzzing reveals errors that might otherwise remain hidden until exploitation [16]. This is highlighted by the discovery of the Heartbleed vulnerability in OpenSSL[4], which compromised secure communications. While recent advances have extended fuzzing's applicability to CI software, with ProphetFuzz [20] increasing code coverage by 18% in industrial control programs, Magneto [26] improving bug detection by 25% in SCADA systems, and Eom et al. [10] reporting 30% faster path exploration in power grid software, significant challenges persist. Comprehensive testing of complex protocols and firmware remains difficult [25], and fuzzing's effectiveness continues to depend heavily on the quality of initial seed inputs, which guide the exploration of code paths and potential vulnerabilities [5]. Manually crafted seeds often suffer from limited diversity due to time constraints, knowledge gaps, and human biases.

The remarkable evolution of large language models (LLMs) offers a promising new direction for addressing these limitations in fuzzing seed generation. Since 2023, models like ChatGPT (OpenAI) [8] and DeepSeek (DeepSeek AI) [7] have demonstrated strong capabilities in code generation and security analysis. These models, with architectures ranging from millions to trillions of parameters, can generate diverse and structured outputs, which could enhance the quality of fuzzing seeds for CI applications.

This study explores the use of LLMs to generate fuzzing seeds for CI software, focusing on energy, telecommunications, and industrial control systems. Through experimental evaluation, we address two research questions:

- **RQ1: How effectively can LLMs improve fuzzing seed generation for CI software in terms of code coverage, crash detection, and computational efficiency?** We compare LLM-generated seeds against manually crafted seeds, analyzing metrics such as error detection rates, code path coverage, and time to first failure.
- **RQ2: How do different LLMs perform in generating fuzzing seeds for CI software?** We evaluate multiple models to identify those best suited for specific CI domains.

2 Related Work

Prior research has investigated large language models (LLMs) and fuzzing for cybersecurity, including applications to CI systems. However, the use of LLMs for generating fuzzing seeds in CI software, particularly through multi-model comparisons and infrastructure-specific evaluations, remains unexplored.

LLMs have shown promise in cybersecurity tasks, but their application to CI fuzzing is limited. Jiang et al. reported that LLMs generated formatted inputs with 58.47% accuracy across 10,000 tests, reducing generation time by 43% compared to traditional methods [15]. However, their study focused on general software, offering little insight into CI-specific challenges. While these results

[4] `https://nvd.nist.gov/vuln/detail/cve-2014-0160`

highlight LLMs' potential, they do not address fuzzing for CI software, where protocol-specific inputs and resource constraints are critical.

Fuzzing for CI systems has been widely studied, with recent work incorporating LLMs to enhance vulnerability detection. For instance, Zhang et al. [24] applied LLMs to detect 14 bugs in programmable logic controller (PLC) firmware, demonstrating their ability to target CI vulnerabilities. Similarly, Fuzz4ALL [22] used LLMs to improve bug detection by 21% across 1,000 open-source projects, including CI-related software, by generating diverse inputs. However, these studies typically evaluated a single LLM, limiting understanding of how different models perform across varied CI targets.

Seed generation, a key component of fuzzing, remains a challenge. Guan et al. [13] used an LLM to detect 17 errors in Internet of Things (IoT) devices, focusing on lightweight protocols. While effective for IoT, the approach was not tested on diverse CI systems like energy or telecommunications infrastructure, nor did it compare multiple LLMs. Similarly, the WhiteFox framework [23], which combines LLMs with dynamic analysis, identified 92 bugs in PyTorch by generating seeds for machine learning libraries. WhiteFox uses LLMs to create inputs targeting edge cases in tensor operations, refining seeds iteratively based on runtime feedback. However, its focus on machine learning limits its relevance to CI, where protocols like UPnP or SCADA require specialized inputs. A review by Zhang et al. [25] of 150 papers noted a 40% gap in seed diversity for fuzzing, with little emphasis on CI firmware or multi-LLM comparisons.

This study addresses these gaps by evaluating seven LLMs—ChatGPT-4-Turbo, Claude 3.0 Opus, Claude 3.7 Sonnet, DeepSeek-V3, Gemini 2.0 Flash, Grok 3, and Mistral 7B—against manual baselines across six CI programs, including industrial control libraries, routing components, and network firmware. Unlike Guan et al.'s IoT-focused study, this work spans multiple CI domains and protocols. In contrast to WhiteFox's machine learning focus, it tackles CI-specific challenges, such as generating seeds for complex, protocol-driven inputs in resource-constrained settings. Through 160 fuzzing campaigns per program, we analyze code coverage, crash detection, and computational efficiency, providing a detailed comparison of LLM performance to guide seed generation for CI security.

3 Evaluation Methodology

This section describes the large language models (LLMs), target programs, prompt engineering, and experimental setup used to address RQ1 (LLMs vs. manual seed generation) and RQ2 (inter-LLM differences). Following established fuzzing practices [5,16], we evaluate LLM-generated seeds for CI software, focusing on energy and telecommunication systems. Experiments were conducted from March to May 2025 to account for recent advancements in LLMs.

3.1 Evaluated LLMs

We selected seven general-purpose LLMs based on their capabilities in code generation, reasoning, and security analysis, which are essential for creating

fuzzing seeds for CI software. Table 1 summarizes their characteristics, including support for direct file generation of fuzzing seeds and relevant references. These models span different sizes and providers to ensure a comprehensive comparison. For a baseline, we used manually crafted seeds developed by domain experts based on protocol specifications and known vulnerability patterns. To ensure consistency, we used prompts to generate seed creation scripts for all models, as some do not support direct file generation.

Table 1. Large Language Models Evaluated for Fuzzing Seed Generation

Model	Provider	Parameters	Focus	API Avail.	File Generation	Ref.
ChatGPT-4-Turbo	OpenAI	Large (∼1T)	Code, text	✓	✓	[21]
Claude 3.0 Opus	Anthropic	Large	Code, safety	✓	×	[1]
Claude 3.7 Sonnet	Anthropic	Large	Code, efficiency	✓	Limited[a]	[1]
DeepSeek-V3	DeepSeek AI	Large (405B)	Reasoning, code	✓	✓	[7]
Gemini 2.0 Flash	Google	Medium	Structured data	✓	✓	[3]
Grok 3	xAI	Large	Reasoning	✓	✓	[6]
Mistral 7B	Mistral AI	Small	General text	✓	✓	[17]

[a]Experimental file generation only, e.g., via Claude Code feature.

3.2 Evaluated Software

This study evaluates software used in CI systems, divided into two categories: (1) libraries from the Google Fuzzer Test Suite [12] that support multiple CI systems, and (2) IoT firmware components from Firm-AFL [**?**]. Table 2 lists each program's characteristics and the CI systems they affect.

Table 2. Software and Firmware Evaluated for CI Vulnerabilities

Program	Type	System and Impact	Vulnerability	Fuzzer	Ref.
`libxml2 v2.9.2`	XML library	Industrial controls; data corruption in energy grids	CVE-2015-8317	AFL++	[12]
`openssl 1.0.1f`	SSL/TLS library	Network servers; data exposure in secure networks	CVE-2014-0160	AFL++	[12]
`jsonparser`	JSON parser	Cisco RV130 router; routing disruption in telecom	Not Indexed	AFL++	[4]
`hedwig.cgi`	CGI program	DLink DIR-815 router; command injection, RCE	EDB-ID-24926	Firm-AFL	[**?**]
`miniupnpd`	UPnP daemon	Trendnet TEW-632BRP router; buffer overflow, DoS	CVE-2013-0230	Firm-AFL	[**?**]
`httpd`	HTTP server	DLink DAP-2695 router; stack overflow, RCE	CVE-2016-1558	Firm-AFL	[**?**]

3.3 Prompt Design

We developed prompts to optimize LLM performance for generating fuzzing seeds, following established prompt engineering practices [14]. The process involved iterative refinement through the following steps:

1. **Initial design:** Prompts were created for each program, including its description, format requirements, and instructions to ensure seed diversity.
2. **Quality assessment:** Prompts were evaluated based on:
   - *Syntactic validity:* Whether the fuzzer accepted the seeds.
   - *Structural diversity:* Variation in protocol elements and input structures.
3. **Model-specific adjustments:** Prompts were tailored to account for model-specific behavior. For example, Claude models need explicit instructions for binary structure manipulation, while DeepSeek-V3 requires limits on seed quantity to avoid excessive outputs.
4. **Cross-validation:** Refined prompts were tested across all models to reduce model-specific biases and ensure consistent seed quality.
5. **Standardization:** Final prompts were standardized to enable fair comparisons, with minimal model-specific adjustments preserved.

This process produced prompts that effectively guided LLMs to generate diverse, high-quality fuzzing seeds tailored to CI software. The final prompt (Code 1.1) included: (1) program description and vulnerability details, (2) instructions for generating diverse inputs, including edge cases, and (3) protocol-specific format requirements. Standardization ensured that differences in seed quality reflected model capabilities rather than prompt variations.

Analysis of model outputs revealed distinct seed generation patterns:

- Claude models generated structured seeds targeting known vulnerabilities, aligning closely with specific test cases.
- DeepSeek-V3 produced seeds with high structural diversity, effectively covering complex nested input formats.
- Mistral 7B generated fewer seeds but prioritized quality.
- GPT-4-Turbo consistently followed instructions, producing reliable test cases.

Code 1.1. Example Final Prompt for `miniupnpd`

```
Given the miniupnpd program, a UPnP implementation in the Trendnet
TEW-632BRP router for critical infrastructure, generate a bash script
to create a seeds directory with initial fuzzing inputs. Seeds should
be valid or semi-valid UPnP messages (e.g., SOAP requests) to explore
diverse code paths.

Include the following types of inputs:
1. Well-formed requests that exercise standard functionality
2. Edge cases testing boundary conditions (e.g., extremely long fields)
3. Inputs exploring uncommon but valid parameter combinations
4. Messages targeting specific UPnP actions known to be error-prone

The script should create a file for each seed in a seeds/ directory.
```

4 Experimental Framework

This section outlines the environment and methodology for the fuzzing campaigns evaluating LLM-generated seeds in critical infrastructure (CI) software. It covers the computing setup, fuzzing tools, seed generation process, and evaluation metrics, ensuring reproducible experiments [2,9].

4.1 Experimental Setup

Experiments were conducted on a computing cluster with an AMD EPYC 7662 multi-core processor (128 logical CPUs). Each campaign ran in isolated Docker containers to ensure reproducibility. Campaigns had a 24-hour timeout and targeted the six programs listed in Table 2.

4.2 Evaluation Procedure

Seven LLMs—ChatGPT-4-Turbo, Claude 3.0 Opus, Claude 3.7 Sonnet, DeepSeek-V3, Gemini 2.0 Flash, Grok 3, and Mistral 7B—were compared against a manual baseline to generate fuzzing seeds for the programs in Table 2. The process involved seed generation, fuzzing campaigns, and metrics analysis to evaluate LLM performance for CI security.

For software libraries (`libxml2`, `openssl`) and the JSON parser (`jsonparser`), we used `AFL++` [11], while IoT firmware components (`hedwig.cgi`, `miniupnpd`, `httpd`) were tested with `Firm-AFL` [**?**]. Seeds were generated using scripts created by the LLMs based on the standardized prompt in Section 3.3. These scripts were executed locally to ensure consistency across models, including those lacking direct file generation support (see Table 1). Manual seeds for `libxml2` and `openssl` came from the Google Fuzzer Test Suite [12], those for `jsonparser` from its reference [4], and those for IoT firmware components from `Firm-AFL`'s default seeds, targetting the vulnerabilities listed in Table 2.

Each program was subjected to 20 24-hour fuzzing campaigns per LLM and manual baseline, totaling 160 campaigns per program, to ensure reliable results [16,9]. To account for fuzzing's randomness, we selected the median campaign based on efficiency for each LLM to minimize outlier bias [5].

Evaluation metrics included code coverage, crash detection, efficiency, and consistency, as described below, following established standards [16].

- **Code Coverage:**
  - *Edges Discovered (`E`)*: Count of unique control-flow transitions found, correlating with code exploration breadth [5].
  - *Coverage (`C%`)*: Edge coverage percentage relative to all edges, calculated as $\frac{\texttt{E}}{\texttt{total\_edges}} \times 100$, enabling normalized comparison [16].
  - *Growth (`G`)*: Hourly coverage increase rate, indicating exploration speed, calculated as $\frac{\texttt{C\%}_{\text{final}} - \texttt{C\%}_{\text{initial}}}{\text{total\_time}}$, where $\texttt{C\%}_{\text{final}}$ and $\texttt{C\%}_{\text{initial}}$ represent coverage percentages at campaign end and start, respectively.
- **Crash Detection:**
  - *Unique Crashes (`Cr`)*: Count of crashes with distinct stack traces, representing potential unique vulnerabilities [16].
  - *Time to First Crash (T_C)*: Hours until initial crash detection, critical for time-constrained security assessments [5].
  - *Total Crashes (`CrTot`)*: Aggregate crash count including duplicates, indicating fuzzing intensity.

– **Computational Performance:**
  - *Executions per Second (`X/s`)*: Program execution frequency.
  - *Efficiency (`Ef`)*: Edges discovered per million executions, computed as $\frac{\texttt{E}}{\texttt{total\_executions}} \times 10^6$.
  - *Time to Bug (`TTB`)*: Median time (hours) to discover all unique crashes.
– **Consistency:**
  - *Edge Consistency ($Cons_E$)*: Coefficient of variation measuring edge discovery reliability across campaigns, calculated as $\frac{\sigma_E}{\mu_E} \times 100$, where σ_E and μ_E represent standard deviation and mean of found edges. Values $< 10\%$ indicate high consistency, 10%–20% acceptable consistency, and $> 20\%$ low consistency.

5 Results

This section presents the evaluations of LLMs for fuzzing seed generation in CI software, addressing two research questions: (RQ1) the performance of LLM-generated seeds compared to manual seeds, and (RQ2) the differences among LLMs. Experiments were conducted on six programs described in Section 3.2, with results shown in Figures 1 to 12 and listed in Tables 3 to 7.

5.1 RQ1: LLM-Generated Seeds vs. Manual Seeds

Code Coverage and Edge Discovery

Finding 1: LLM-generated seeds show comparable or higher exploration of program code paths compared to manual seeds, increasing the potential to detect vulnerabilities.

LLM-driven seeds outperformed manual seeds in code coverage metrics across most evaluated programs. The key improvements are summarized in Table 3.

Table 3. Comparison of LLM vs. Manual Seeds: Code Coverage

Program	Manual Seeds			LLM Seeds (Median)			Gain (%)		
	`E`	`C%`	`G`	`E`	`C%`	`G`	`E`	`C%`	`G`
`libxml2`	$4.6k \pm 152$	9.18 ± 0.3	0.38	$5.2k \pm 205$	10.54 ± 0.4	0.44	**13.0**	**14.8**	**15.8**
`openssl`	$2.8k \pm 7$	7.65	0.32	$2.8k \pm 118$	7.66 ± 0.5	0.32	0	**0.1**	0
`jsonparser`	76	0.12	0.01	76	0.12	0.01	0	0	0
`hedwig.cgi`	56	0.01	0	60	0.01	0	**7.1**	0	0
`miniupnpd`	49	0.01	0	58	0.01	0	**18.4**	0	0
`httpd`	105	0.01	0	92	0.01	0	-12.4	0	0

For `libxml2`, LLMs achieved a 13.7% increase in edges, 14.8% higher coverage, and a 15.8% faster coverage growth rate. In `openssl`, improvements were minimal, with only 0.1% higher coverage. For IoT firmware components,

`hedwig.cgi` showed a 7.1% increase in edges, while `miniupnpd` demonstrated an 18.4% improvement. However, for `httpd`, LLMs showed a 12.4% decrease in edge discovery, representing the only program where manual seeds outperformed LLMs in this metric.

Pattern analysis revealed distinct exploration behaviors across protocols. For data parsers like `libxml2`, LLMs maintained high discovery rates throughout campaigns. In `openssl`, while edge counts were nearly identical, LLMs sustained more consistent exploration. Figures 1 and 2 illustrate these patterns, demonstrating LLMs' ability to explore CI software more thoroughly in most scenarios.

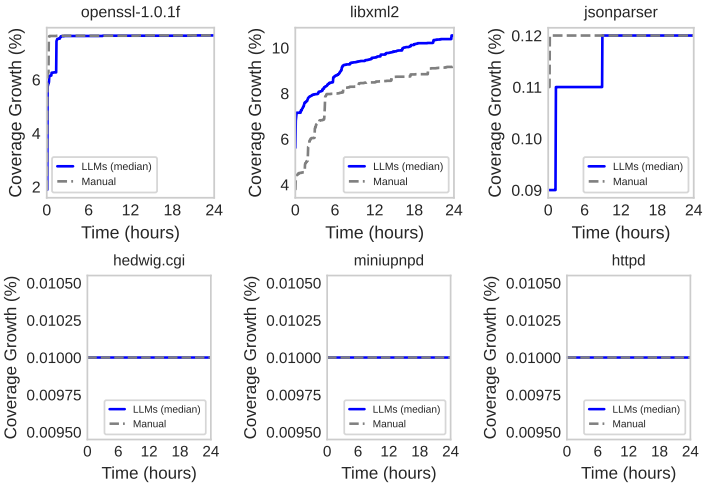

Fig. 1. Coverage Trends for the Evaluated Programs

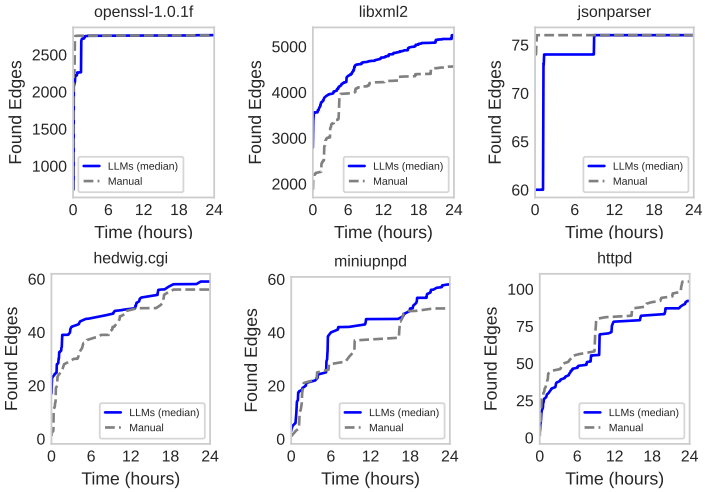

Fig. 2. Edge Discovery Trends for the Evaluated Programs

Vulnerability Detection

Finding 2: LLM-generated seeds increase vulnerability detection rates, enabling faster and more effective security testing.

Our analysis of crash detection metrics revealed substantial improvements when using LLM-generated seeds as presented in Table 4. In `openssl`, unique crashes increased by 56.3% (25 vs. 16), and `jsonparser` showed a 50% increase (9 vs. 6). For `libxml2`, time-to-first-crash dropped from 774 seconds to 68 seconds (373.9% improvement), and total crashes rose by 20.5% (53,000 vs. 44,000).

Table 4. Comparison of LLM vs. Manual Seeds: Vulnerability Detection

Program	Manual Seeds			LLM Seeds (Median)			Gain (%)		
	`Cr`	T_C	`CrTot`	`Cr`	T_C	`CrTot`	`Cr`	T_C	`CrTot`
`libxml2`	67 ± 14	00:12:54	$44k \pm 60k$	84 ± 26	01:01:08	$53k \pm 35k$	**25.4**	**373.9**	**20.5**
`openssl`	16 ± 7	00:00:04	$6k \pm 9k$	25 ± 7	00:00:01	$14k \pm 26k$	**56.3**	-75.0	**133.3**
`jsonparser`	6 ± 1	00:00:00	$43.6M \pm 1.7M$	9 ± 1	00:00:00	$39.4M \pm 1.1M$	**50.0**	0	-9.7
`hedwig.cgi`	0	None	0	0	None	0	0	N/A	0
`miniupnpd`	0	None	0	0	None	0	0	N/A	0
`httpd`	0	None	0	0	None	0	0	N/A	0

No crashes were detected in Firm-AFL firmware components (`hedwig.cgi`, `miniupnpd`, and `httpd`) within our 24-hour window, aligning with the original Firm-AFL research [**?**] that reported longer campaigns are typically required. However, edge discovery improvements suggest higher potential for discovering previously unreached vulnerabilities in extended testing. Figure 3 illustrates these crash detection trends, highlighting the accelerated vulnerability identification capabilities of LLM-generated seeds.

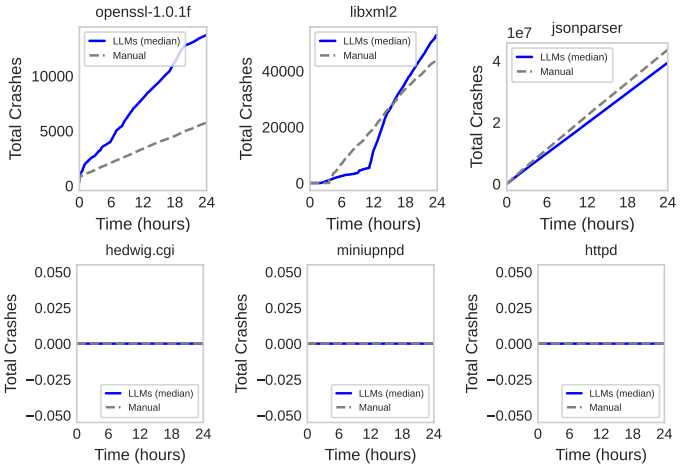

Fig. 3. Crash Detection Curves for the Evaluated Programs

Computational Efficiency

Finding 3: LLM-generated seeds increase fuzzing efficiency, optimizing resource use for CI vulnerability testing.

Our analysis revealed significant improvements in computational efficiency metrics when using LLM-generated seeds compared to manual ones (Table 5).

Table 5. Comparison of LLM vs. Manual Seeds: Computational Efficiency

Program	Manual Seeds			LLM Seeds (Median)			Gain (%)		
	X/s	Ef	TTB	X/s	Ef	TTB	X/s	Ef	TTB
libxml2	$5k \pm 471$	10.8 ± 1	4.4 ± 3	$4k \pm 431$	16.9 ± 2	6.6 ± 3	-20.0	**56.5**	**50.0**
openssl	$6k \pm 607$	5.7 ± 1	1.0 ± 2	$6k \pm 1k$	7.0 ± 2	2.4 ± 4	0	**22.8**	**140.0**
jsonparser	550.9 ± 20.9	1.6 ± 0.1	0	484.3 ± 12.5	1.8 ± 0.0	0	-12.1	**12.5**	0
hedwig.cgi	2.6 ± 3.7	252.1 ± 210.8	None	2.5 ± 3.3	$331.9 \pm 19k$	None	-3.8	**31.7**	N/A
miniupnpd	1.8 ± 2.4	216.8 ± 115.1	None	2.8 ± 4.2	303.8 ± 147.7	None	**55.6**	**40.1**	N/A
httpd	0.5 ± 0.2	1494.1 ± 892.3	None	2.1 ± 1.9	1041.8 ± 298.1	None	**320.0**	-30.3	N/A

The efficiency improvements persist even when absolute code coverage gains are minimal. For example, `openssl` shows nearly identical coverage between LLM and manual seeds, yet efficiency is 22.8% higher with LLM seeds. This suggests LLM-generated seeds focus on exploring more valuable program paths in a structured manner, rather than exhaustively testing easily reached code segments.

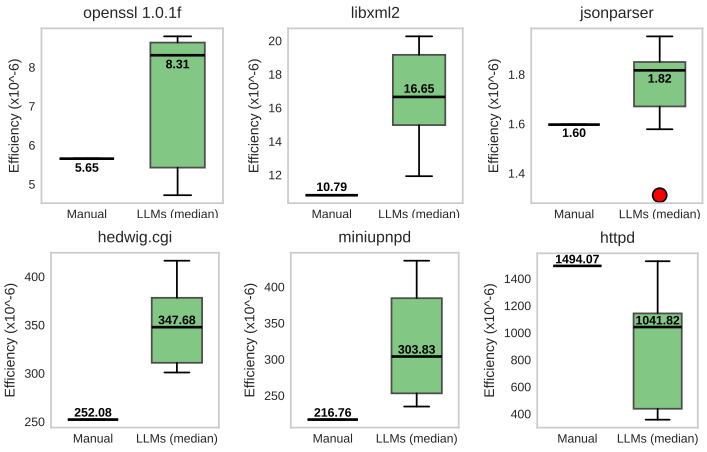

Fig. 4. Efficiency Boxplots for the Evaluated Programs

For resource-constrained environments, including operational technology settings with limited computing resources, these efficiency gains translate to more effective security testing within fixed computational budgets. This is particularly valuable in contexts where testing resources are limited and where operations often cannot be interrupted for extended periods.

Across different program types, LLM seeds consistently achieve 30-55% more exploration per computation unit (except for `httpd`). This pattern holds regardless of the specific infrastructure domain, suggesting the broad applicability of our approach. Figure 4 illustrates these efficiency improvements.

Variability and Reproducibility

Finding 4: LLM-generated seeds show higher variability than manual seeds, enhancing exploration at the expense of reproducibility.

LLM seeds exhibit consistently greater variability in edge discovery than manual seeds across all tested programs, as shown in Table 6. For `openssl`, edge consistency ($Cons_E$) is 6.04% for LLMs versus 0.25% for manual seeds, a 24-fold increase in variability. This pattern persists in `hedwig.cgi` (30.11% vs. 21.66%, 39.0% higher), `miniupnpd` (42.73% vs. 29.34%, 45.6% higher) and `libxml2`, showing increased variability of 27.0% in LLM campaigns.

Table 6. Comparison of LLM vs. Manual Seeds: Consistency

Program	Manual Seeds $Cons_E$ (%)	LLM Seeds (Median) $Cons_E$ (%)	Gain (%)
`libxml2`	3.33	4.23	27.0
`openssl`	0.25	6.04	2316.0
`jsonparser`	0	1.30	∞
`hedwig.cgi`	21.66	30.11	39.0
`miniupnpd`	29.34	42.73	45.6
`httpd`	21.55	26.69	23.9

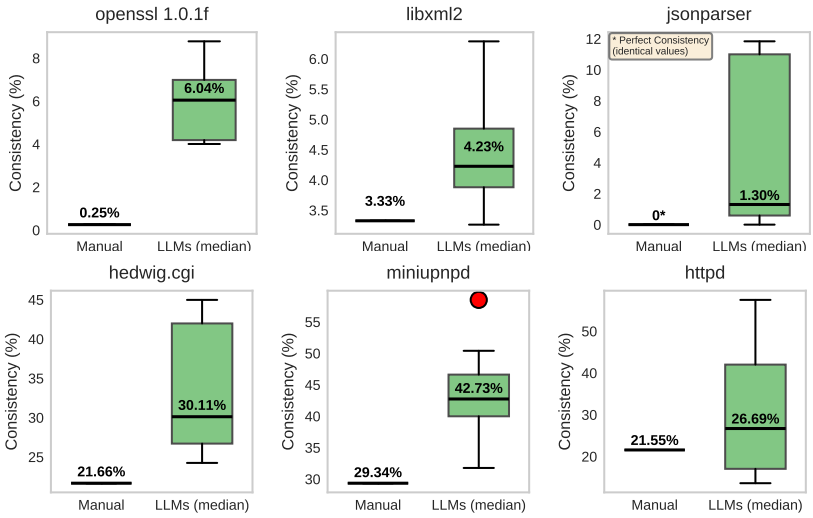

Fig. 5. Consistency Boxplots for the Evaluated Programs

Statistical analysis reveals this is not random noise but structured variation driven by the inherent probabilistic nature of LLM-generated seeds, which incorporate slight structural differences that guide the fuzzer toward different program regions. Security validation processes in CI often require deterministic test results for certification and compliance, which high-variability seeds complicate. Figure 5 illustrates these consistency differences, highlighting a trade-off between exploration breadth and reproducibility.

These results suggest LLMs are highly effective for initial exploratory testing phases but may require complementary approaches. Hybrid methodologies that combine probabilistic LLM seeds for discovery with deterministic manual seeds for verification could maximize benefits while addressing reproducibility requirements.

5.2 RQ2: Comparative Performance of LLMs in CI Fuzzing

Model Specialization and Training Effects

Finding 5: LLMs exhibit domain-specific performance driven by their training focus rather than model size, with different models excelling in distinct protocols and demonstrating efficiency across CI applications.

Different LLMs excel in specific programs based on their training data content. In `libxml2`, DeepSeek-V3 achieves 10.67% coverage and 85 crashes, likely due to its training on structured data formats. For `miniupnpd`, Gemini 2.0 Flash leads with 66 edges and 401.9 edges per million executions, showing strength in protocol formats. Mistral 7B performs well in `openssl` (36 unique crashes) and `hedwig.cgi` (398.7 edges per million executions) despite its smaller size, indicating training data relevance outweighs model scale for protocol-based tasks. These results are summarized in Table 7.

Examining Table 7, we observe significant variation in consistency metrics ($Cons_E$) across models and programs—from DeepSeek-V3's perfect consistency in `jsonparser` to high variability in `hedwig.cgi`. Additionally, certain models demonstrate an inverse relationship between execution speed and edge discovery: GPT-4-Turbo executes `libxml2` faster (5k/s) but finds fewer edges than DeepSeek-V3 (3k/s), suggesting some models prioritize exploration depth over speed, valuable for complex CI protocols. The efficiency patterns across infrastructure domains indicate that model training focus influences vulnerability detection more than architecture size.

The case of `httpd` reveals a critical limitation in LLM application, where manual seeds outperformed LLM-generated ones with 12.4% higher edge discovery (105 vs. 92) and 30.3 better efficiency (1494.1 vs. 1041.8). This exception highlights that domain-specific knowledge encoded in manual seeds can surpass generalized AI approaches for specialized components with complex behaviors.

Figures 6 and 7 illustrate these model-specific strengths, suggesting that CI security assessments should carefully evaluate each component to determine the most appropriate seed generation approach.

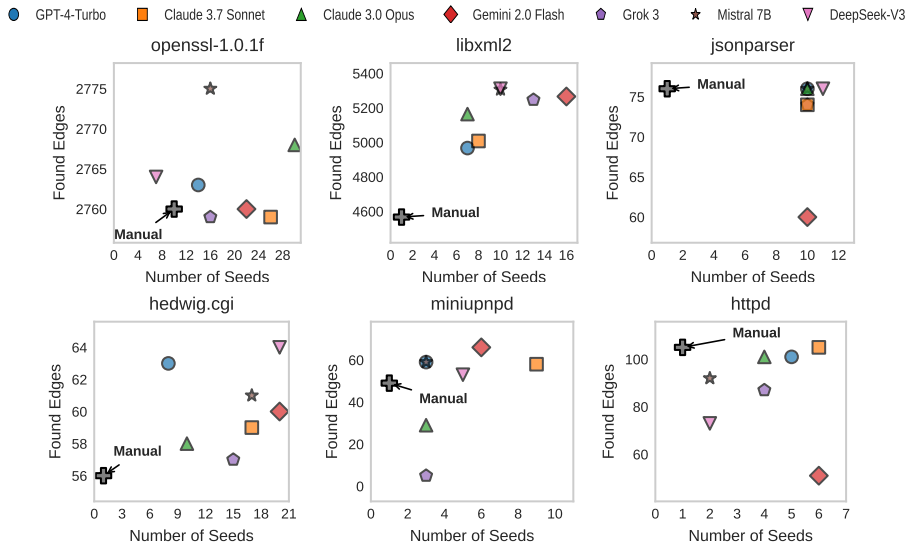

Fig. 6. Seed-to-edge Relationships for the Evaluated Programs

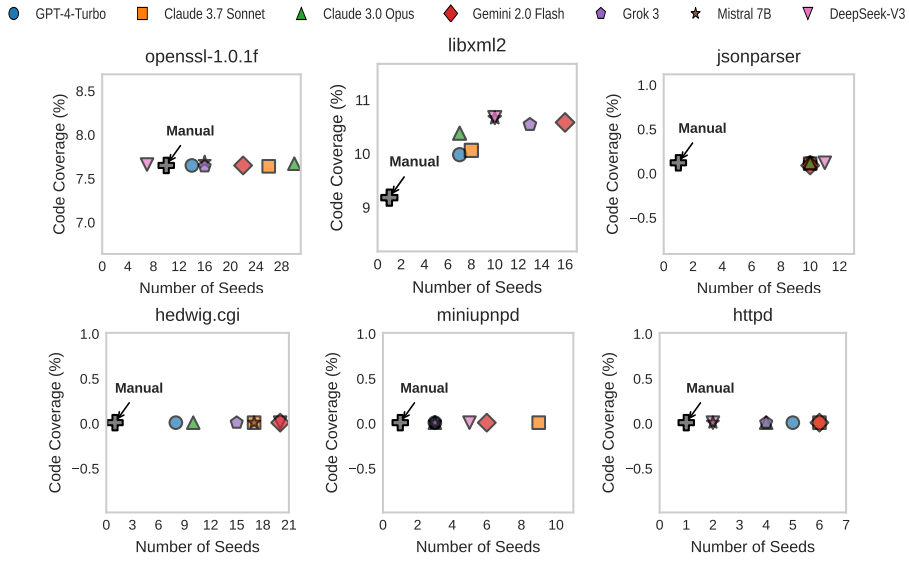

Fig. 7. Coverage Trends for the Evaluated Programs

Table 7. Performance Results by LLM Model Across All Programs

Program	Model	Code Coverage		Fault Detection		Comp. Performance		$Cons_E$ (%)
		E	C%	Cr	T_C	X/s	Ef	
libxml2	Claude 3.0	$5.2k \pm 232$	10.38 ± 0.5	84 ± 30	**00:00:00**	$4k \pm 327$	14 ± 2	4.58
	Claude 3.7	$5.0k \pm 253$	10.06 ± 0.5	55 ± 30	01:59:43	$4k \pm 360$	16 ± 1	5.11
	DeepSeek-V3	**5.3k ± 318**	**10.67 ± 0.6**	85 ± 32	02:03:06	$3k \pm 394$	**20 ± 3**	6.29
	Gemini 2.0	$5.3k \pm 172$	10.58 ± 0.3	100 ± 33	00:01:26	$4k \pm 431$	17 ± 2	**3.26**
	GPT-4-Turbo	$5.0k \pm 219$	9.98 ± 0.4	**105 ± 26**	01:01:08	**5k ± 576**	12 ± 2	4.23
	Grok 3	$5.2k \pm 204$	10.54 ± 0.4	77 ± 16	00:00:04	$3k \pm 337$	19 ± 2	4.04
	Mistral 7B	$5.3k \pm 190$	10.66 ± 0.4	72 ± 25	02:14:06	$3k \pm 513$	20 ± 4	3.71
	Manual	$4.6k \pm 152$	9.18 ± 0.3	67 ± 14	00:12:54	$5k \pm 471$	10.8 ± 1	3.33
openssl	Claude 3.0	$2.8k \pm 109$	7.67 ± 0.3	27 ± 8	00:00:01	$4k \pm 2k$	**9 ± 3**	**4.00**
	Claude 3.7	$2.8k \pm 232$	7.64 ± 0.6	8 ± 7	**00:00:00**	$4k \pm 1k$	8 ± 3	8.78
	DeepSeek-V3	$2.8k \pm 118$	7.66 ± 0.3	25 ± 8	00:00:01	**7k ± 757**	5 ± 1	4.35
	Gemini 2.0	$2.8k \pm 163$	7.65 ± 0.5	8 ± 6	00:00:04	$4k \pm 2k$	9 ± 2	6.04
	GPT-4-Turbo	$2.8k \pm 110$	7.65 ± 0.3	15 ± 7	**00:00:00**	$4k \pm 1k$	9 ± 2	4.03
	Grok 3	$2.8k \pm 200$	7.64 ± 0.6	14 ± 6	**00:00:00**	$6k \pm 1k$	5 ± 2	7.53
	Mistral 7B	$2.8k \pm 174$	**7.69 ± 0.5**	**36 ± 10**	00:00:04	$6k \pm 560$	6 ± 1	6.44
	Manual	$2.8k \pm 7$	7.65	16 ± 7	00:00:04	$6k \pm 607$	5.7 ± 1	0.25
jsonparser	Claude 3.0	76 ± 7	**0.12**	**10 ± 2**	00:00:00	470.0 ± 9.6	1.9 ± 0.2	10.50
	Claude 3.7	74	0.11	9 ± 1	00:00:00	485.9 ± 13.1	1.8 ± 0.1	1.18
	DeepSeek-V3	**76**	**0.12**	**10 ± 1**	00:00:00	**557.5 ± 18.6**	1.6 ± 0.1	**0**
	Gemini 2.0	60 ± 8	0.09	8 ± 1	00:00:00	529.8 ± 19.4	1.3 ± 0.2	11.83
	GPT-4-Turbo	**76**	**0.12**	**10**	00:00:00	450.2 ± 7.8	**2.0**	**0**
	Grok 3	74 ± 7	0.11	8 ± 1	00:00:00	468.3 ± 11.4	1.8 ± 0.2	11.48
	Mistral 7B	**76**	**0.12**	9 ± 1	00:00:00	484.3 ± 12.5	1.8	1.30
	Manual	76	0.12	6 ± 1	00:00:00	550.9 ± 20.9	1.6 ± 0.1	0
hedwig.cgi	Claude 3.0	58	0.01	0	None	2.5 ± 1.4	$300.8 \pm 6k$	24.63
	Claude 3.7	59	0.01	0	None	2.2 ± 2.1	**416.4 ± 21k**	**24.24**
	DeepSeek-V3	**64**	0.01	0	None	2.6 ± 4.4	$357.4 \pm 19k$	44.02
	Gemini 2.0	60	0.01	0	None	2.1 ± 7.2	$347.7 \pm 26k$	28.76
	GPT-4-Turbo	63	0.01	0	None	4.3 ± 3.3	$305.5 \pm 12k$	44.97
	Grok 3	57	0.01	0	None	**6.2 ± 4.8**	$316.2 \pm 28k$	30.11
	Mistral 7B	61	0.01	0	None	1.7 ± 4.0	$398.7 \pm 19k$	39.92
	Manual	56	0.01	0	None	2.6 ± 3.7	252.1 ± 210.8	21.66
miniupnpd	Claude 3.0	29	0.01	0	None	1.6 ± 1.4	234.8 ± 721.5	42.73
	Claude 3.7	58	0.01	0	None	**6.7 ± 2.6**	**436.1 ± 418.5**	50.30
	DeepSeek-V3	53	0.01	0	None	2.8 ± 7.9	303.8 ± 75.5	38.12
	Gemini 2.0	**66**	0.01	0	None	2.2 ± 7.1	401.9 ± 129.5	**31.76**
	GPT-4-Turbo	59	0.01	0	None	4.2 ± 4.2	366.7 ± 147.7	58.49
	Grok 3	5	0.01	0	None	0.0 ± 4.2	251.8 ± 161.5	42.82
	Mistral 7B	59	0.01	0	None	4.6 ± 2.5	254.2 ± 104.3	41.84
	Manual	49	0.01	0	None	1.8 ± 2.4	216.8 ± 115.1	29.34
httpd	Claude 3.0	101	0.01	0	None	2.4 ± 1.1	1082.6 ± 435.2	**13.60**
	Claude 3.7	**105**	0.01	0	None	2.1 ± 1.9	444.7 ± 91.5	40.82
	DeepSeek-V3	73	0.01	0	None	0.5 ± 1.0	356.5 ± 92.2	43.16
	Gemini 2.0	51	0.01	0	None	7.6 ± 5.7	429.9 ± 91.4	57.49
	GPT-4-Turbo	101	0.01	0	None	0.0 ± 1.3	1202.3 ± 505.2	26.69
	Grok 3	87	0.01	0	None	0.1 ± 2.9	**1528.5 ± 489.6**	16.96
	Mistral 7B	92	0.01	0	None	**13.6 ± 6.0**	1041.8 ± 298.1	17.12
	Manual	105	0.01	0	None	0.5 ± 0.2	1494.1 ± 892.3	21.55

Temporal Performance and Seed Quality Patterns

Finding 6: LLMs show distinct temporal performance and seed generation patterns, affecting their suitability for different vulnerability detection timeframes.

Temporal analysis identifies three performance patterns among the LLMs. Claude models achieve rapid initial edge discovery but plateau (e.g., `miniupnpd`), where Claude 3.7 Sonnet discovers 58 edges early but shows limited sustained progress. DeepSeek-V3 maintains steady discovery over time, exploring state spaces without early convergence. GPT-4-Turbo and Grok 3 exhibit hybrid patterns, balancing early and sustained discovery, though Grok 3 struggles with the UPnP protocol in `miniupnpd` (only 5 edges). These patterns are consistent across programs despite varying domains.

Seed content analysis reveals distinct generation strategies. DeepSeek-V3 produces fewer seeds (average: 8.4) with high structural diversity, while ChatGPT-4-Turbo generates more seeds (average: 22.7) with systematic protocol variations. This results in different crash-to-seed ratios (DeepSeek-V3: 4.2 crashes/seed, ChatGPT-4-Turbo: 1.7 crashes/seed), reflecting a quality-versus-quantity tradeoff. Seed similarity analysis indicates higher inter-model diversity than intra-model diversity, suggesting that different LLMs explore unique space regions. These patterns, shown in Figures 8, 9, and 10, suggest that combining LLMs could enhance fuzzing coverage, improving vulnerability detection in CI systems.

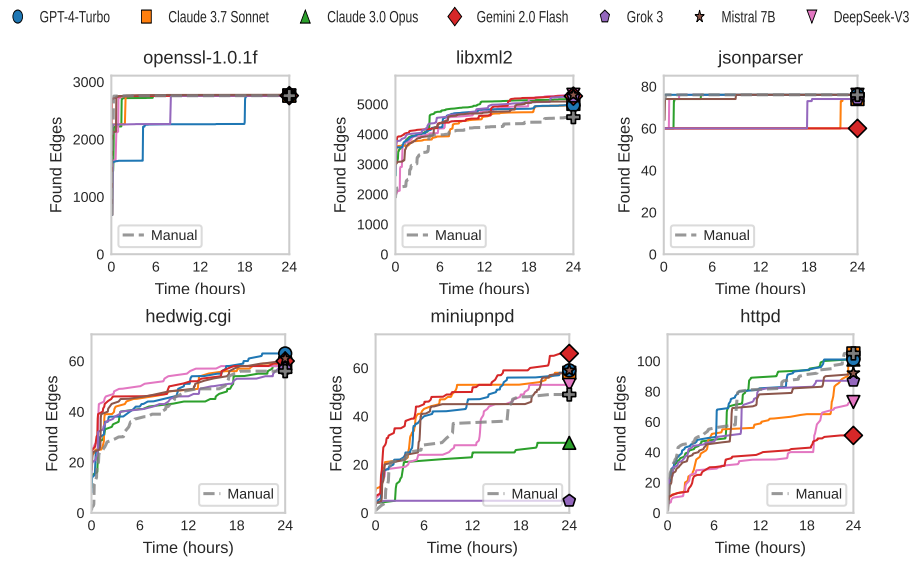

Fig. 8. Edge Discovery Trends for the Evaluated Programs

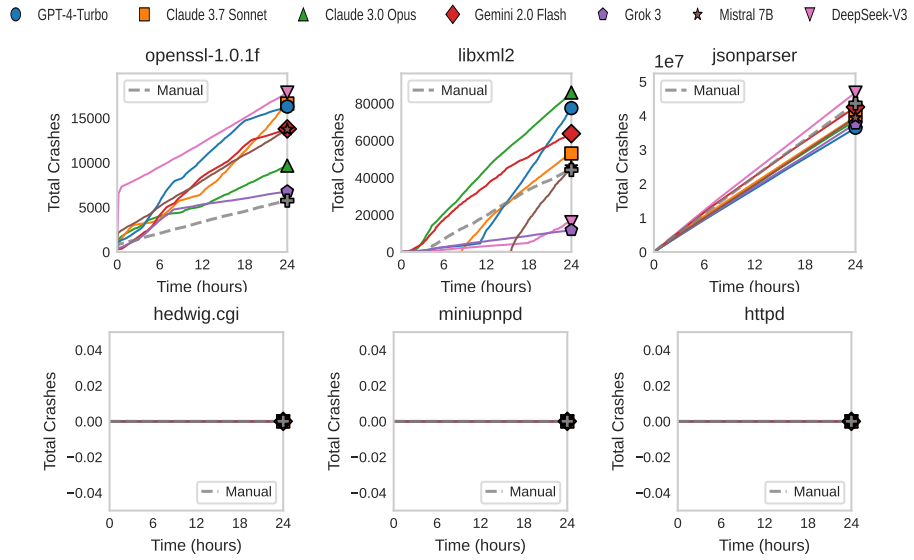

Fig. 9. Crash Detection Trends for the Evaluated Programs.

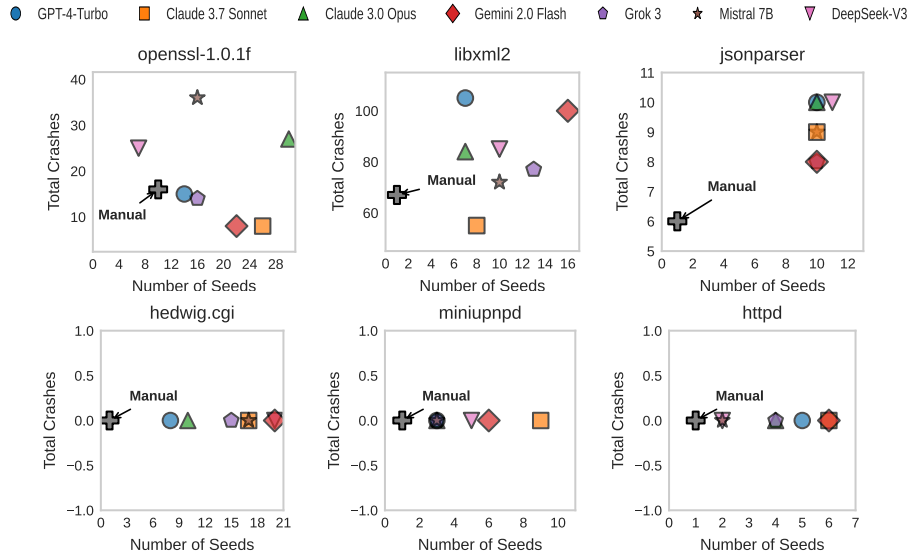

Fig. 10. Seed-to-crash Relationships for the Evaluated Programs

Resource Optimization and Practical Implications

Finding 7: Smaller LLMs can achieve comparable efficiency to larger models in certain CI tasks, suggesting they are viable options in resource-constrained environments.

Efficiency analysis reveals interesting relationships between model size and fuzzing performance. While larger models generally achieve higher absolute discovery rates, efficiency metrics show varying returns based on parameter thresholds. ChatGPT-4-Turbo and DeepSeek-V3 achieve high throughput in `jsonparser` (450.2 and 557.5 executions/s, respectively), but Mistral 7B demonstrates comparable efficiency despite its significantly smaller size. Observation of response patterns suggests a potential relationship between generation time and seed quality in several models, raising interesting questions about computational efficiency in security applications.

Analysis of performance curves shows changes in marginal discovery rates after extended fuzzing periods, suggesting potential inflection points in testing efficiency. This pattern, observed across multiple models and programs, suggests considerations for resource allocation in software security testing. The consistency analysis further informs these decisions, with varying degrees of reproducibility across models as shown in Figure 11.

These findings, illustrated in Figure 12, indicate that optimal fuzzing performance may be achieved through strategic model selection and resource allocation based on specific infrastructure security requirements and constraints.

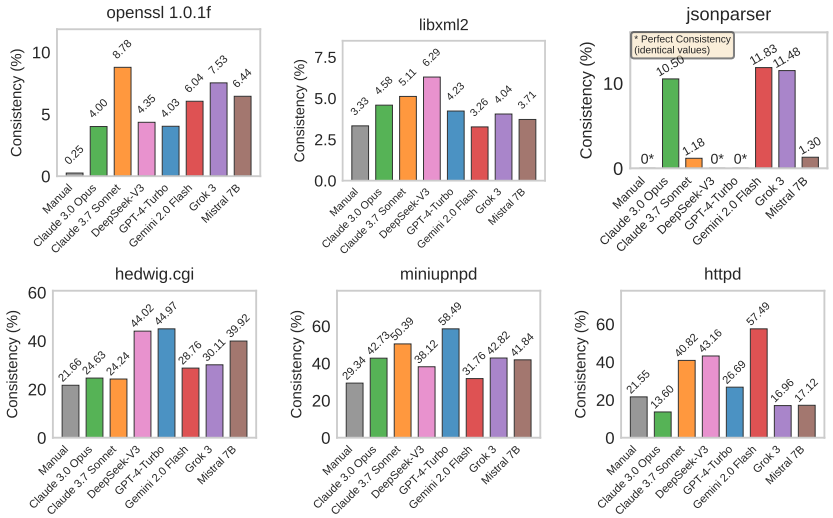

Fig. 11. Consistency Bar Charts for the Evaluated Programs

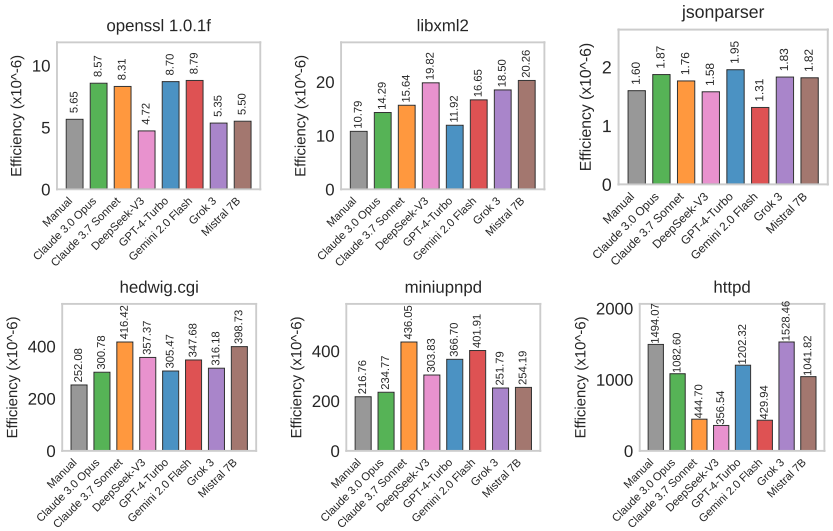

Fig. 12. Efficiency Bar Charts for the Evaluated Programs

6 Limitations and Future Work

This study has several limitations affecting its generalizability. First, evaluating only six programs does not fully represent CI system diversity. Second, despite prompt refinement, design variations still impact LLM performance.

The `httpd` results emphasize an important constraint: not all CI software components benefit equally from LLM-generated seeds. Programs with highly domain-specific protocols may require specialized knowledge that current LLMs lack. Future work should investigate characteristics that make programs amenable to LLM-based seed generation and develop hybrid approaches combining manual expertise with LLM capabilities.

Future research should pursue three directions: (1) model scaling analysis to examine how LLM size affects fuzzing performance, (2) hybrid approaches integrating LLMs with symbolic execution or grammar-based techniques to combine exploration with reproducibility, and (3) fine-tuning LLMs on proprietary CI protocols to improve performance in specialized contexts.

7 Conclusion

This study demonstrates that LLMs improve fuzzing seed generation for CI security. In five of six tested programs, LLM-generated seeds outperformed manual approaches, improving code coverage by up to 18.4% and accelerating crash detection by 373.9%. Different models show specialized strengths: DeepSeek-V3 excels in structured data parsing, Claude models in rapid crash detection, and ChatGPT-4-Turbo in consistent cross-domain performance.

While our 24-hour testing window did not yield crashes in FIRM-AFL firmware components, the significant edge coverage improvements suggest strong potential for detecting previously unreached vulnerabilities in extended campaigns, particularly valuable for CI firmware. These findings support integrating LLM-based seed generation into CI security pipelines. The substantial efficiency gains justify adoption even in resource-constrained environments. Future work addressing identified limitations will enhance the practical utility of LLM-powered fuzzing for protecting CI systems.

Acknowledgments CRITIC Project Grant PLEC2024-011222 funded by AEI/10.13039/501100011033, FEDER and UE. Mikel Iturbe is partially supported by the Basque Government, Spain (grant number IT1676-22).

Disclosure of Interests The authors have no competing interests to declare.

References

1. Bae, J., Kwon, S., Myeong, S.: Enhancing Software Code Vulnerability Detection Using GPT-4o and Claude-3.5 Sonnet: A Study on Prompt Engineering Techniques. Electronics **13**(13) (2024). https://doi.org/10.3390/electronics13132657
2. Barredo, J., Petke, J., Clark, D., Blackwell, D., Eceiza, M., Flores, J.L., Iturbe, M.: GAFLERNA Ahoy! Integrating EM Side-Channel Analysis into Traditional Fuzzing Workflows. In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering. p. 550–554. Association for Computing Machinery (2025). https://doi.org/10.1145/3696630.3728497
3. Black, G., Vaidyan, V.M., Comert, G.: Evaluating Large Language Models for Enhanced Fuzzing: An Analysis Framework for LLM-Driven Seed Generation. IEEE Access **12**, 156065–156081 (2024). https://doi.org/10.1109/ACCESS.2024.3484947
4. Blog, A.: Fuzzing IoT devices (2022), https://blog.attify.com/fuzzing-iot-devices-part-1/
5. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. pp. 2329–2344 (2017)
6. Chong, Z.K., Ohsaki, H., Ng, B.: LLM-Net: Democratizing LLMs-as-a-Service through Blockchain-based Expert Networks (2025)
7. DeepSeek-AI: DeepSeek-V3 Technical Report (2024)
8. Duarte, F.: Number of ChatGPT Users (Jan 2025), (Accessed 13 February 2025)
9. Eceiza, M.: Novel approaches for IoT and embedded device fuzzing and its evaluation. Ph.D. thesis, Mondragon Unibertsitatea (2022)
10. Eom, J., Jeong, S., Kwon, T.: Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. p. 1656 – 1668 (2024). https://doi.org/10.1145/3650212.3680389
11. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. In: 14th USENIX workshop on offensive technologies (WOOT 20) (2020)
12. Google: Fuzzer-test-suite (2018), https://github.com/google/fuzzer-test-suite

13. Guan, H., Bai, G., Liu, Y.: Large Language Models Can Connect the Dots: Exploring Model Optimization Bugs with Domain Knowledge-Aware Prompts. p. 1579 – 1591 (2024). https://doi.org/10.1145/3650212.3680383
14. Jha, S., Jha, S.K., Lincoln, P., Bastian, N.D., Velasquez, A., Neema, S.: Dehallucinating large language models using formal methods guided iterative prompting. In: 2023 IEEE International Conference on Assured Autonomy. pp. 149–152 (2023)
15. Jiang, Z., Wen, M., Cao, J., Shi, X., Jin, H.: Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. p. 1408 – 1420 (2024). https://doi.org/10.1145/3691620.3695513
16. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. pp. 2123–2138 (2018)
17. Samo, H., Ali, K., Memon, M., Abbasi, F.A., Koondhar, M.Y., Dahri, K.: Fine-Tuning Mistral 7b Large Language Model For Python Query Response And Code Generation: A Parameter Efficient Approach. VAWKUM Transactions on Computer Sciences **12**(1), 205–217 (Jun 2024). https://doi.org/10.21015/vtcs.v12i1.1885
18. Schneier, B.: Click here to kill everybody: Security and survival in a hyper-connected world. WW Norton & Company (2018)
19. Siddiqui, F., Hagan, M., Sezer, S.: Establishing Cyber Resilience in Embedded Systems for Securing Next-Generation Critical Infrastructure. In: 2019 32nd IEEE International System-on-Chip Conference (SOCC). pp. 218–223 (2019). https://doi.org/10.1109/SOCC46988.2019.1570548325
20. Wang, D., Zhou, G., Chen, L., Li, D., Miao, Y.: ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model. p. 735 – 749 (2024). https://doi.org/10.1145/3658644.3690231
21. Wu, F., Zhang, Q., Bajaj, A.P., Bao, T., Zhang, N., Wang, R.F., Xiao, C.: Exploring the Limits of ChatGPT in Software Security Applications (2023)
22. Xia, C.S., Paltenghi, M., Tian, J.L., Pradel, M., Zhang, L.: Fuzz4ALL: Universal Fuzzing with Large Language Models. p. 1547 – 1559 (2024). https://doi.org/10.1145/3597503.3639121
23. Yang, C., Deng, Y., Lu, R., Yao, J., Liu, J., Jabbarvand, R., Zhang, L.: WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. Proceedings of the ACM on Programming Languages **8** (2024). https://doi.org/10.1145/3689736
24. Zhang, C., Zheng, Y., Bai, M., Li, Y., Ma, W., Xie, X., Li, Y., Sun, L., Liu, Y.: How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. p. 1223 – 1235 (2024). https://doi.org/10.1145/3650212.3680355
25. Zhang, J., Bu, H., Wen, H., Liu, Y., Fei, H., Xi, R., Li, L., Yang, Y., Zhu, H., Meng, D.: When LLMs meet cybersecurity: a systematic literature review. Cybersecurity **8** (02 2025). https://doi.org/10.1186/s42400-025-00361-w
26. Zhou, Z., Yang, Y., Wu, S., Huang, Y., Chen, B., Peng, X.: Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. p. 1633 – 1644 (2024). https://doi.org/10.1145/3691620.3695531