LLM-Guided Fuzzing for Pathological Input Generation

Didier Ishimwe[0000−0001−8470−3835] and Thanhvu Nguyen[0000−0002−4255−4592]

George Mason University, Fairfax VA 22030, USA
{dishimwe,tvn}@gmu.edu

Abstract. Pathological inputs can trigger worst-case algorithmic behavior, leading to excessive resource consumption and revealing performance bottlenecks. Generating such inputs is difficult because they follow highly specific patterns that random testing or traditional fuzzing rarely uncovers. Existing fuzzing techniques either rely on domain-specific knowledge, which limits generality, or apply mutations on binary representations of inputs that produce predominantly invalid inputs.

We present `PathoGen`, a feedback-driven fuzzing framework that integrates LLMs into evolutionary search to efficiently discover pathological inputs. By leveraging LLMs for candidate generation, `PathoGen` achieves high-quality input without requiring domain-specific knowledge, such as handcrafted grammars. An adaptive prompt template mechanism ensures robustness during input generation, while convergence tracking by input size directs computational resources toward those most likely to expose worst-case behaviors.

Evaluation across textbook algorithms and real-world applications shows that `PathoGen` uncovers expected theoretical complexity bounds, induces significant slowdowns in practical systems, and outperforms existing domain-independent fuzzers in efficiency and input validity while performing comparably to domain-specific fuzzers within their specialized domains.

Keywords: Complexity Analysis · Performance Testing · Fuzzing · Large Language Models

1 Introduction

Reasoning about a program's worst-case complexity is crucial for applications such as performance analysis, optimization validation, and security assessment [3,14]. Because programs can exhibit vastly different resource usage on typical versus worst-case inputs, generating *pathological inputs* (those that trigger worst-case algorithmic behaviors [12]) is essential for accurate complexity analysis. These inputs help validate theoretical bounds, reveal performance bottlenecks, expose excessive resource usage [6], and highlight potential DoS attack vectors [15,33].

Pathological inputs often follow highly specific patterns that traditional random or mutation-based fuzzing is unlikely to discover, and they can reveal previously unanticipated worst-case runtime behaviors that are costly and difficult

to detect. They typically manifest as nonfunctional defects, causing excessive resource consumption rather than outright crashes, leading to silent failures that escape conventional detection. To mitigate these challenges, companies including Amazon have adopted automated complexity analysis to catch performance regressions early in development [22,32]. Nevertheless, systematically identifying pathological inputs remains a difficult problem, as programs are complex and may rely on inaccessible library code, while pathological inputs vary widely across domains without a universal pattern [26].

To address the challenge of identifying worst-case performance behaviors, prior work has explored two main directions: automated complexity analysis and input generation. Automated complexity analyses aim to formally infer asymptotic bounds on resource usage [4,10]. While these approaches provide sound bounds, they do not generate concrete inputs that both witness the tightness of the bound and help developers debug unexpected performance issues [26].

In contrast, automated input generation, uncovers inputs that maximize resource consumption during execution. These techniques operate by exploring the input space through fuzzing [12] or synthesis [26,28], using runtime feedback as a guide to identify inputs that trigger worst-case behaviors. Some focus on domain-specific strategies tailored to structured inputs [13,15,28], while others aim for domain-independent solutions that can apply across a range of applications [20].

Existing input generation techniques face notable limitations. Domain-specific approaches achieve high precision by relying on program analysis and other domain-specific knowledge to guide exploration. For example, they may use handcrafted grammars [28], code vulnerability patterns discovered through static analysis [2,14,29], or symbolic execution to explore program paths [17]. However, these approaches restrict the programming languages and input formats that they can support. In contrast, domain-independent techniques apply mutations to byte-level input representations [12,20]. Such byte-level manipulation often produces a high proportion of invalid inputs (up to 80% for regular expressions [20]), wastes computational resources, and ignores program semantics, thereby limiting the discovery of deep performance bottlenecks that depend on subtle control-flow patterns [12].

In this paper, we present `PathoGen` (Pathological Input Generation), a fuzzing approach for discovering inputs that trigger worst-case resource usage. `PathoGen` enhances evolutionary fuzzing with Large Language Models (LLMs), allowing it to generate syntactically valid, diverse inputs without relying on handcrafted grammars or binary encodings. Input candidates are generated, executed, and scored by resource cost, with high performers guiding subsequent generations. Input generation is driven by autoprompting [30], where LLM prompts initialized with user-provided input specifications (e.g., natural language description) are iteratively refined using runtime feedback. `PathoGen` introduces per-size convergence tracking: it simultaneously generates pathological inputs for a set of input sizes, monitoring whether further generations at each size improve resource usage scores. Once a size converges, it is pruned from the search space. This allows smaller sizes to converge early while exploration continues for larger ones,

eliminating wasteful search and scaling efficiently to realistic input lengths. The campaign terminates when every considered size converges or the budget is exhausted.

We evaluate `PathoGen` on a broad suite of benchmarks and demonstrate its effectiveness in generating pathological inputs for both classical algorithms and real-world applications. For textbook algorithms, `PathoGen` consistently produces inputs that realize the expected worst-case complexity. Compared to domain-independent complexity fuzzers such as `Slowfuzz` [20], `PathoGen` discovers pathological inputs more quickly and with substantially fewer invalid cases. At the same time, it narrows the performance gap with domain-specific fuzzers by achieving comparable slowdowns in their target domains, including regular expressions (`Revealer` [15]) and Java programs (`Acquirer` [14]), thus bringing domain-independent fuzzing much closer to domain-specific approaches in practice. Overall, these results highlight `PathoGen`'s advancement of domain-independent performance fuzzing through LLM-guided, feedback-driven input generation.

In summary, this paper makes the following contributions:

- We present `PathoGen`, a domain-independent fuzzer that integrates LLMs into a feedback-driven evolutionary loop, eliminating the need for handcrafted grammars or binary encodings. `PathoGen` is publicly available at `https://github.com/dynaroars/pathogen`
- A novel per-size convergence mechanism for efficiently discovering pathological inputs across input sizes.
- An extensive evaluation showing that `PathoGen` surpasses existing domain-independent fuzzers in efficiency and input validity, while achieving performance comparable to domain-specific fuzzers on both classical algorithms and real-world applications.

2 Overview

2.1 Large Language Models

Recent advances in machine learning have driven the widespread adoption of LLMs for both natural language and code-related tasks. Instruction-tuned LLMs, such as GPT-4, demonstrate strong abilities to follow complex instructions, reason over structured text, and understand code [16]. LLMs are adapted to tasks through two main paradigms: fine-tuning, which updates model weights using task data but demands significant resources, and prompting, which steers model outputs through carefully designed inputs [21]. Prompting has become the dominant, lightweight approach, with autoprompting [30] iteratively refining prompts for better output.

LLMs have shown promise in software engineering tasks such as program synthesis [25], testing [1,8,23,30,31], and formal methods [9]. `PathoGen` builds on this trend by embedding LLMs directly into the fuzzing loop, replacing traditional mutation and crossover operators with the goal of maximizing resource usage.

This integration leverages LLMs emergent structural and semantic awareness to generate higher-quality inputs more efficiently, expanding applicability, reducing invalid or redundant cases, and improving discovery of worst-case behaviors compared to mutation-based fuzzers.

2.2 Illustrative Example: Quicksort

```
def qsort(arr):
    if len(arr) <= 1:
        return arr
    p = arr[0]
    l, r= [], []
    for x in arr[1:]:
        if x < p:
            l.append(x)
        else:
            r.append(x)
    return qsort(l) + [p] + qsort(r)
```

(a) **Quicksort implementation**

Fuzzing progress tracking

Iter	Active Sizes	Batch	Status
1	[4, 7, 16, 32]	16	All active
5	[7, 16, 32]	12	4: converged
12	[16, 32]	8	8: converged
18	[32]	4	16: converged
25	[ ]	0	All converged

Per-size evolution example

Size 4	Best Input	Score
Iter 1	[3, 1, 2, 5]	32
Iter 2	[1, 2, 5, 3]	45
Iter 3-5	[1, 2, 3, 5]	45

(b) **per-size convergence tracking**

System: Generate pathological inputs to maximize instruction count, focusing on constructs that increase runtime (e.g., nested loops, recursion, edge cases).
Target Program: `{program_code}`
Input Format: `{input_description}`
Size: `{size_description}`
Output: Raw inputs only, one per line; no explanations or extra text.

User: Generate `{batch_size}` inputs for sizes `{target_sizes}`, producing `{k}` inputs per size.
Strictly follow the input format above. Each input should aim to outperform the current best example at its size in terms of instruction count when executed on the target program
Current Best: `{best_inputs}`
Output: Raw inputs only, one per line; no explanations or extra text.

(c) **LLM prompt structure**

Fig. 1: Illustrative example of pathological input generation for `quicksort`

To demonstrate our approach, we use the widely studied `quicksort` algorithm, a standard benchmark in performance and complexity analysis [4,10,14,17,28] due to its simple control flow, basic data types, and well-understood semantics. It offers a clean setting to isolate performance behaviors without the confounding influence of complex control structures or data representations.

Identifying pathological inputs for `quicksort` is challenging due to its sensitivity to implementation details, such as pivot selection. In our implementation (Fig. 1a), choosing the first element as the pivot yields worst-case $O(n^2)$ behavior. Despite its simplicity, several complexity analysis tools report incorrect

or overly conservative bounds: `TiML`[27] infers $O(n^3)$, `Chora`[4] reports $O(n2^n)$, and `Dynaplex`[10] incorrectly infers $O(n)$. By contrast, recent pathological input generation tools produce inputs that trigger the $O(n^2)$ worst-case behavior: `Singularity`[28], using handcrafted grammars, and `Acquirer`[14], guided by statically detected vulnerable code patterns.

Motivated by the need to systematically discover pathological inputs across a range of sizes, we frame the problem as a per-size search: given a program, the goal is to identify inputs of varying sizes that induce high resource consumption, such as instruction count. To address this, `PathoGen` employs an adaptive evolutionary fuzzing strategy guided by LLMs, simultaneously exploring multiple input sizes. For a range of sizes, `PathoGen` tracks convergence per size and removes those that reach optimal resource usage, focusing computation on sizes with remaining improvement potential. The algorithm terminates when all sizes have converged or the fuzzing budget is exhausted.

Let us use Fig. 1 to illustrate how `PathoGen` discovers pathological inputs for the `quicksort` example. The fuzzing campaign begins with a system prompt (Fig. 1c). The LLM's system prompt captures information that persists across iterations, such as the programs source code (Fig. 1a), the expected input specification, the meaning of input *size*, the objective to maximize instruction count, and the expected LLM output format. The input specification can range from a natural language description (e.g., list of integers) to a more complex Python object. The more precise the input specification, the better the LLM can generate valid inputs. At each iteration, `PathoGen` uses a user prompt (Fig. 1c) to instantiate multiple target sizes, the batch size (e.g, 16 inputs), and the number of inputs per size ($k = 4$). Each size maintains independent state tracking with the current best score and convergence monitoring, as shown in Fig. 1b.

As the campaign progresses, sizes that fail to improve after a threshold number of iterations are removed from the active set. In this example, size 4 failed to improve its score from iteration 2 to 5, leading `PathoGen` to consider it converged, reducing the batch size from 16 to 12 inputs. The LLM user prompt (Fig. 1c) is continuously updated with the best performing inputs and their scores for each active size, enabling targeted refinement. This process continues until all sizes converge to their optimal instruction counts or until the fuzzing budget is exhausted, systematically discovering pathological inputs across the size spectrum. This performance-driven feedback loop allows `PathoGen` to efficiently generate pathological inputs.

3 Proposed Approach

`PathoGen` combines evolutionary fuzzing with LLM-guided input generation to discover pathological inputs that trigger worst-case performance behaviors in programs. Within the evolutionary framework, the LLM replaces traditional crossover and mutation operations, generating candidates guided by concrete resource usage metrics. This LLM-driven generation allows convergence to be reached faster, but at different rates for different input sizes. To avoid wasteful

computation on sizes that have already converged, `PathoGen` introduces per-size adaptive convergence tracking, which simultaneously monitors multiple input sizes and dynamically reallocates computational effort by pruning sizes that have reached optimal performance. The remainder of this section details the system architecture and its core components, including the fuzzing loop, prompt structure, input evaluation, and candidate refinement strategy.

We frame the pathological input generation as a per-size optimization problem: given a program and a resource metric, the goal is to find inputs of varying sizes that maximize the programs' measured resource usage under that metric. Because the input space is too large for exhaustive exploration, we adopt an adaptive evolutionary strategy in which an LLM generates candidate inputs for multiple target sizes simultaneously, guided by dynamically structured prompts and performance feedback. As sizes converge to their optimal resource usage, they are pruned from the search space, concentrating computation on sizes with remaining improvement potential. The system requires three user-provided inputs: (i) the target program, given as source code or an executable, (ii) an input specification defining format and size, and (iii) a resource metric such as instruction count.

3.1 Prompt Architecture

The effectiveness of LLM-guided input generation depends critically on well-structured prompts that provide clear guidance, context, and feedback to the LLM. `PathoGen` employs a multi-template prompt system tailored to different phases of the fuzzing workflow. This subsection outlines the prompt architecture, input specification, and template parameterization that enable systematic pathological input discovery.

`PathoGen` uses three prompt templates, each serving a distinct role:

1. *System prompt*: establishes persistent context and behavioral guidelines across iterations.
2. *Generation prompt*: provides iteration-specific objectives and feedback.
3. *Error-recovery prompt*: handles LLM induced failures and guides corrective behavior.

System Prompt. The system prompt defines the foundational context shared across all fuzzing iterations. It includes the target program, the fuzzing objective, the input specification, precise input size semantics with examples, and output formatting rules. The prompt also highlights algorithmic complexity triggers, such as nested loops, recursion depth, and costly edge cases, to steer the LLM toward pathological inputs rather than arbitrary ones.

Generation Prompt. The generation prompt encodes iteration-specific instructions and performance feedback. It specifies the number of inputs to produce (batch size), distributes them across active target sizes, and includes the best-performing inputs per size with their scores. By explicitly requiring the LLM to outperform current best examples *at the same size*, the prompt enforces competitive refinement without allowing trivial escalation through larger inputs.

Error-Recovery Prompt. When an input fails validation, `PathoGen` issues a recovery prompt containing the rejected examples and error details. This corrective feedback reinforces input requirements and supplies both valid and invalid examples, enabling the LLM to adapt its generation strategy and reduce repeated errors. Although modern LLMs follow instructions reliably, they may still produce invalid or unexpected outputs, making structured prompts essential for robustness.

Input Specification. Users provide structured input specifications in `YAML`. Each specification defines: (i) a description of the input domain and constraints (natural language or code), (ii) a size-calculation method (e.g., list length or byte count), and (iii) valid and invalid examples. For example, a quicksort specification defines inputs as "comma-separated integers enclosed in square brackets, with `[5, 2, 8, 1, 9]` as a valid example and `"2, 3, 5"` as invalid". The specification also includes a size function (e.g., Python code), ensuring precise mapping between inputs and target sizes. Explicit size definitions are crucial to avoid ambiguity in per-size optimization, especially since algorithmic complexity analysis often depends on different notions of *size* even for similar input domains. For instance, Karatsuba multiplication and Fibonacci programs both take integers as input, but their complexity sizes correspond to the number of digits and the integer value, respectively [7].

Template Parameterization. All prompts support dynamic parameterization through runtime placeholders, such as `target_sizes` in Fig. 1c for coordinating per-size search and `num_inputs` in Fig. 1c for adaptive batch sizing. Because templates are parameterized in this way, users can modify them to tailor prompts to specific domains, e.g., by supplying pathological patterns relevant to the program under analysis, while still providing the same runtime parameters. This flexibility allows the same template structure to support diverse input domains, from integer lists in sorting algorithms to class definitions, while maintaining consistent semantics and feedback.

3.2 Fuzzing Algorithm

`PathoGen` proceeds in four phases: initialization, evaluation, iterative fuzzing with error handling, and termination, as summarized in Algorithm 1.

Initialization. The algorithm begins by initializing the active size set, best seeds, error and convergence trackers, and prompt (line 1). Active sizes are spaced using an inverse-logarithmic scale to emphasize medium to large inputs, where algorithmic complexity behaviors are more likely to emerge. For each size, trackers monitor the best resource usage and consecutive non-improving iterations to detect convergence. Error tracking enables graceful recovery from invalid LLM outputs, and the seeds retain the best inputs per size for subsequent iterations.

Evaluation. During each iteration, candidate inputs are generated for all active sizes (line 3) and evaluated according to the specified resource metric (line 4). Valid candidates are mapped to their sizes, with per-size scores and errors tracked to enable independent selection and feedback.

Algorithm 1 `PathoGen`: Pathological Input Generation

Require: Program (P), $InputSpec$, Resource metric (R), $Budget$
1: $Seeds$, $Trackers$, $ActiveSizes$, $Prompts \leftarrow$ initialize_state(P, $InputSpec$, R)
2: **while** $Budget > 0$ **and** $ActiveSizes \neq \emptyset$ **do**
3: $Candidates \leftarrow$ generate_candidates($ActiveSizes$)
4: $Scores, Trackers \leftarrow$ evaluate_candidates($Candidates$, P, R)
5: $State \leftarrow$ update_state($Scores$, $Seeds$, $Trackers$, $ActiveSizes$)
6: $Prompt \leftarrow$ update_prompt($State$)
7: **if** $Errors$ **then**
8: $Prompts \leftarrow$ recovery_prompts($Trackers$)
9: **end if**
10: $Convergence \leftarrow$ update_budget($ActiveSizes$, $Budget$)
11: **end while**
12: **return** $Seeds$

Fitness Scoring and Selection. Within each size class, `PathoGen` updates the stored seeds only if a candidate achieves a higher resource usage score (line 5). This size-based competition prevents trivial improvements via larger inputs and encourages the discovery of costly structural patterns. Trackers are updated each iteration, resetting on improvement and incrementing otherwise to monitor convergence.

Prompt Update and Error Handling. After updating the state, prompts are regenerated to incorporate performance feedback (line 6). If candidates fail validation, error-recovery prompts with prior errors as feedback are issued (line 8), guiding the LLM to produce valid inputs in subsequent iterations. This ensures that computation is not wasted on repeated invalid outputs.

Convergence and Termination. Sizes that have converged to optimal resource usage are pruned from the active set, focusing effort on remaining active sizes (line 10). The loop continues until either the budget is exhausted or all sizes have converged. Finally, the algorithm returns the best inputs per size, representing the discovered pathological inputs (line 12).

4 Evaluation

To assess the effectiveness of our approach, we implemented it in a prototype tool called `PathoGen`. This section outlines the benchmark design and describes our evaluation methodology.

Benchmarks We evaluate `PathoGen` on a benchmark suite compiled from prior work on performance fuzzing and complexity analysis, including `Revealer`[15], `Acquirer`[14], and `WISE` [5]. The benchmarks cover domains such as regular expression parsing, sorting algorithms, and graph algorithms. For complexity-bound inference, we selected programs with well-defined worst-case behaviors to provide theoretical bounds as ground truth. We also included programs from real-world software, such as Java's regex parser.

The benchmark suite was curated to evaluate `PathoGen`'s capabilities in two key aspects: first, whether it can generate inputs that achieve the theoretical worst-case complexity for a given implementation; and second, how its performance compares with state-of-the-art pathological input generators in producing slowdowns in real-world applications.

Setups Experiments were conducted on a Debian 12 system with an AMD Ryzen Threadripper PRO 3995WX 64-core CPU, 125GB RAM, 497GB NVMe SSD, and an NVIDIA GeForce RTX 5090 GPU, using the OpenAI `GPT-4-turbo` [19] and Ollama `gpt-oss:20b` [18] models. Each experiment was repeated five times, with median results reported.

4.1 Asymptotic Bound Analysis

We evaluate `PathoGen`'s ability to rediscover known worst-case complexity bounds across a diverse set of classical algorithms, including sorting, data structure operations, and graph algorithms with well-established theoretical asymptotics. The benchmarks are drawn from the `WISE` suite [5], which has also been used by `Acquirer` [14] to evaluate complexity bound discovery.

Program	**Worst**	**Found**
Sorted Linked-List	$O(n^2)$	✓
Heap Insert	$O(n \log n)$	✓
Red-Black Tree	$O(n \log n)$	✓
Quicksort	$O(n^2)$	✓
Binary Search Tree	$O(n^2)$	✓
Merge Sort	$O(n \log n)$	✓
Bellman-Ford	$O(n^3)$	✓
Dijkstra's	$O(n^2)$	✓
Traveling Salesman	$O(n!)$	✓
Insertion Sort	$O(n^2)$	✓

Tab. 1: Evaluation on textbook algorithms.

For each algorithm, we run `PathoGen` and record the maximum instruction count observed per size. For each algorithm, we compare the observed performance against the expected theoretical worst-case performance using linear regression. A bound is considered validated if the observed data aligns with the theoretical complexity class with $R^2 > 0.90$. As summarized in Tab. 1, `PathoGen` successfully recovers the correct asymptotic complexity for all benchmarks, demonstrating its ability to discover different complexity classes for programs with diverse input structures.

4.2 Comparison to State-of-the-Art

We evaluate `PathoGen` against state-of-the-art tools spanning both domain-independent and domain-specific categories of pathological input generation. On the domain-independent side, `SlowFuzz`[20] serves as the primary baseline, as few tools are designed to discover algorithmic complexity vulnerabilities without domain knowledge. We also include `Singularity`[28], which is only partially domain-independent: while it generalizes across programs, it relies on manually crafted grammars to ensure valid input generation. In contrast, the domain-specific space is richer, with many approaches tailored to particular

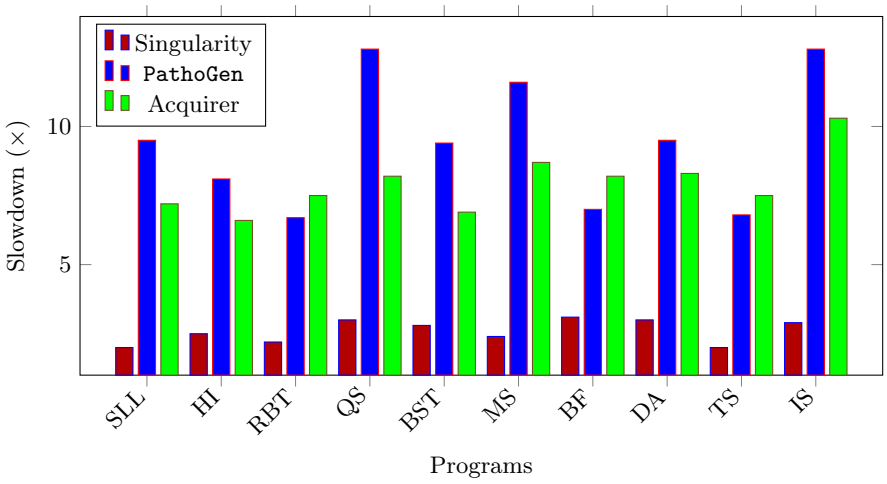

Fig. 2: Slowdown factor comparison of `PathoGen`, `Singularity` and `Acquirer` across textbook algorithms.

formats or vulnerability classes. From this category, we select representative tools: `Acquirer`[14], which uses manually crafted vulnerability patterns to guide search in Java programs, and `Revealer`[15], which targets regex engines through crafted attack strings. Slowdown is measured relative to randomly generated inputs as a baseline, with a thirty-minute timeout per benchmark for all tools.

Our results highlight the difference between domain-informed approaches and naive exploration. `PathoGen`, `Acquirer`, and `Revealer` achieve significant slowdowns in less than 10 minutes on average, whereas `SlowFuzz` and `Singularity`, which rely on byte-level or grammar-based mutations without deeper semantic guidance, often require hours to reach comparable results. `Acquirer` leverages statically identified vulnerable code patterns, while `Revealer` exploits regex-specific models based on nondeterministic finite automata (NFA). In contrast, `PathoGen` uses LLMs to generate high-quality inputs rapidly, without relying on handcrafted grammars or domain-specific knowledge.

Input validity further differentiates the tools. `SlowFuzz` produces up to 80% invalid regex inputs, whereas `PathoGen` maintains at most 15% invalid inputs, depending on the LLM used. OpenAI `GPT-4-turbo` achieves the highest precision, and the results reported here use OpenAI `GPT-4-turbo`. Other models, such as Ollama `gpt-oss:20b`, generate less precise inputs yet still surpass `SlowFuzz` in both efficiency and input validity, while outperforming `Singularity` in efficiency.

Program-Level Slowdowns. Fig. 2 compares `PathoGen`, `Singularity`, and `Acquirer` across 10 textbook algorithms. `PathoGen` consistently outperforms `Singularity`, achieving up to 380% higher slowdowns. `Singularity`'s reliance on handcrafted grammars and naive exploration requires substantially more time to converge, often needing up to three hours to reach slowdowns that `PathoGen` achieves in just ten minutes. While `Acquirer` sometimes produces slightly higher

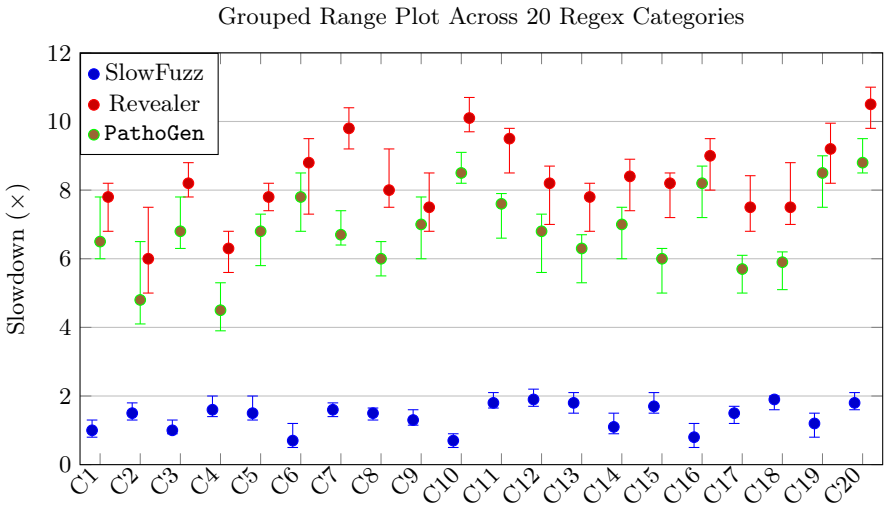

Fig. 3: Grouped range plot showing min, median, and max slowdown for `SlowFuzz`, `PathoGen`, and `Revealer` across 20 regex categories.

slowdowns for certain graph algorithms, `PathoGen` surpasses it in other cases, particularly sorting, demonstrating competitive performance without relying on static analysis or pre-encoded vulnerability patterns. Moreover, since we do not analyze the Java regex engine source code but instead measure its performance via lightweight API calls, `Acquirer`'s statically detected vulnerability patterns are ineffective in this domain.

Regex-Level Slowdowns. Fig. 3 shows performance across 200 regular expression (regex) patterns taken from the benchmark used to evaluate `Revealer`'s algorithmic vulnerability detection. These regexes have also been used to evaluate other tools such as `ReScue` [24]. The full `Revealer` benchmark contains many more regexes across 20 different categories; we randomly selected 10 regexes per category because `PathoGen`'s LLM-guided generation is computationally expensive, making the evaluation of the entire dataset cost-prohibitive. Nonetheless, 200 regexes are sufficient to produce meaningful results. In this experiment, we use the Java 17 built-in regex parser, demonstrating that `PathoGen` can handle practical applications.

`Revealer` consistently achieves higher slowdowns than `PathoGen`, reflecting its domain-specific optimizations based on modeling NFA used in Java's regex parsing engine. In contrast, `SlowFuzz` performs substantially worse due to unguided input mutations and our limited fuzzing budget, requiring up to three hours to generate significant results. Additionally, `SlowFuzz` mutates binary encodings of the regexes to achieve domain independence; however, this approach produces mostly invalid inputs. `PathoGen`'s domain-independent approach achieves performance close to `Revealer` while maintaining broad applicability.

Across all benchmarks, `PathoGen` demonstrates the effectiveness of LLM-guided fuzzing in balancing efficiency, input validity, and generality. It outperforms domain-independent fuzzers and matches or slightly trails domain-specific tools in their targeted areas, illustrating that LLMs can replace handcrafted grammars or vulnerability patterns while still generating competitive pathological inputs.

5 Discussion

`PathoGen` advances domain-independent pathological input fuzzing by producing high-quality, slowdown-inducing inputs without handcrafted grammars, while maintaining strong validity and broad applicability. Key trade-offs remain for practical deployment and future work.

LLM Dependency. Leveraging LLMs offers both benefits and challenges. Large models such as GPT-4 provide high-quality inputs across domains without handcrafted grammars or vulnerability patterns, though they can be costly or access-restricted. Smaller open-source models like Ollama `llama3:7b` are more accessible but less effective for complex inputs. Crucially, as LLM capabilities improve, `PathoGen` automatically benefits from enhanced input generation, whereas domain-specific fuzzers require manual extensions with additional grammars or patterns to achieve similar gains.

Domain Generalization. `PathoGen` relies on LLMs to generate structured inputs, which can lead to lower-quality outputs for uncommon or specialized formats not well represented in the model's training data. However, as companies increasingly adopt LLMs in their production environments, these models are often finetuned on the organization's codebase and typical input formats. When `PathoGen` uses such a finetuned LLM for performance testing on the same codebase, it can generate increasingly effective pathological inputs, surpassing traditional fuzzers that rely on manually crafted grammars or domain-specific patterns. This approach allows `PathoGen` to improve automatically with the adoption and refinement of LLMs, while maintaining broad applicability across applications.

6 Related Work

Fuzzers for pathological inputs fall into two categories: domain-independent and domain-specific. `SlowFuzz`[20] pioneered the domain-independent approach by mutating inputs at the byte level to maximize instruction counts, later extended by `PerfFuzz`[12] with multi-dimensional feedback to better explore hot paths. These methods generalize across formats but often yield invalid inputs, wasting resources.

In contrast, `Singularity` [28] adopts a partially domain-independent strategy by relying on manually crafted grammars to produce valid, pathological inputs via program synthesis across different application domains. `Acquirer` [14] adopts

a hybrid analysis that statically identifies vulnerable code patterns and guides evolutionary search towards program paths with such patterns. A prominent line of work within domain-specific fuzzing targets Denial-of-Service (DoS) vulnerabilities, particularly regex engines where crafted inputs cause excessive backtracking. `Revealer`[15] combines static analysis with dynamic testing, modeling regexes as extended NFA to generate attack strings, while `HotFuzz`[3] applies micro-fuzzing at the method level, using genetic algorithms to evolve inputs that maximize execution time in Java programs. Though effective, these tools are tailored to domains with well-characterized vulnerability patterns and do not generalize to broader algorithmic complexity issues.

Recent advances have explored leveraging LLMs to enhance fuzzing across a variety of domains. Techniques such as `Fuzz4All`[30] improve input diversity, `TELPA`[31] increase coverage, `TitanFuzz`[8] target deep learning libraries, and `BENCIGENS`[1] advance combinatorial testing. Other approaches focus on automated unit test generation [11,23]. To our knowledge, `PathoGen` is the first LLM-based fuzzer explicitly designed for pathological input generation, bridging the gap between domain-independent generality and domain-specific performance optimization.

7 Conclusion

We introduce `PathoGen`, an LLM-guided evolutionary fuzzing framework for discovering inputs that trigger worst-case performance. `PathoGen` bridges the gap between domain-specific precision and domain-independent applicability, enabling scalable pathological input generation across languages and input formats.

For future work, we plan to extend `PathoGen` to support additional user-defined resources such as memory usage and API call patterns, in addition to runtime performance. We also aim to leverage the structural understanding capabilities of LLMs to uncover recurring patterns in discovered pathological inputs. This insight will improve debugging and help developers optimize performance more effectively.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This material is based in part upon work supported by the National Science Foundation under grant numbers 2422036, 2319131, 2238133, and 2200621, and by an Amazon Research Award.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Arcaini, P., Bombarda, A., Gargantini, A.: A search-based benchmark generator for constrained combinatorial testing models. In: 2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 278–287. IEEE (2025)

2. Berglund, M., Drewes, F., Van Der Merwe, B.: Analyzing catastrophic backtracking behavior in practical regular expression matching. arXiv preprint arXiv:1405.5599 (2014)
3. Blair, W., Mambretti, A., Arshad, S., Weissbacher, M., Robertson, W., Kirda, E., Egele, M.: Hotfuzz: Discovering temporal and spatial denial-of-service vulnerabilities through guided micro-fuzzing. ACM Transactions on Privacy and Security **25**(4), 1–35 (2022)
4. Breck, J., Cyphert, J., Kincaid, Z., Reps, T.: Templates and recurrences: Better together. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 688–702. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020)
5. Burnim, J., Juvekar, S., Sen, K.: Wise: Automated test generation for worst-case complexity. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 463–473. IEEE (2009)
6. Carter, K., Ho, S.M.G., Larsen, M.M.A., Sundman, M., Kirkeby, M.H.: Energy and time complexity for sorting algorithms in java. arXiv preprint arXiv:2311.07298 (2023)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
8. Deng, Y., Xia, C.S., Peng, H., Yang, C., Zhang, L.: Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis. pp. 423–435 (2023)
9. Doan, L., Nguyen, T.: Ai-assisted autoformalization of combinatorics problems in proof assistants. In: 2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). pp. 1–5. IEEE (2025)
10. Ishimwe, D., Nguyen, K., Nguyen, T.: Dynaplex: analyzing program complexity using dynamically inferred recurrence relations. Proc. ACM Program. Lang. **5**(OOPSLA), 1–23 (2021)
11. Lemieux, C., Inala, J.P., Lahiri, S.K., Sen, S.: Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 919–931. IEEE (2023)
12. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: Automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 254–265 (2018)
13. Li, Y., Sun, Y., Xu, Z., Cao, J., Li, Y., Li, R., Chen, H., Cheung, S.C., Liu, Y., Xiao, Y.: {RegexScalpel}: Regular expression denial of service ({{{{{ReDoS}}}}}) defense by {Localize-and-Fix}. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 4183–4200 (2022)
14. Liu, Y., Meng, W.: Acquirer: A hybrid approach to detecting algorithmic complexity vulnerabilities. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2071–2084 (2022)
15. Liu, Y., Zhang, M., Meng, W.: Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1468–1484. IEEE (2021)
16. Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., Myers, B.: Using an llm to help with code understanding. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13 (2024)

17. Noller, Y., Kersten, R., Păsăreanu, C.S.: Badger: complexity analysis with fuzzing and symbolic execution. In: International Symposium on Software Testing and Analysis. pp. 322–332 (2018)
18. Ollama: Gpt-oss:20b (2025), https://ollama.com/library/gpt-oss%3A20b
19. OpenAI: Gpt-4 technical report. CoRR **abs/2303.08774** (2023), https://doi.org/10.48550/arXiv.2303.08774
20. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2155–2168 (2017)
21. Pornprasit, C., Tantithamthavorn, C.: Fine-tuning and prompt engineering for large language models-based code review automation. Information and Software Technology **175**, 107523 (2024)
22. Raimondi, F., Chang, B.Y.E.: Calculating the differential cost of code changes (Jun 2022), https://www.amazon.science/blog/calculating-the-differential-cost-of-code-changes
23. Schäfer, M., Nadi, S., Eghbali, A., Tip, F.: Adaptive test generation using a large language model. arXiv preprint arXiv:2302.06527 (2023)
24. Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X., Lu, J.: Rescue: crafting regular expression dos attacks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 225–235 (2018)
25. Sobania, D., Briesch, M., Hanna, C., Petke, J.: An analysis of the automatic bug fixing performance of chatgpt. In: 2023 IEEE/ACM International Workshop on Automated Program Repair (APR). pp. 23–30. IEEE (2023)
26. Wang, D., Hoffmann, J.: Type-guided worst-case input generation. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
27. Wang, P., Wang, D., Chlipala, A.: Timl: a functional language for practical complexity analysis with invariants. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 79 (2017)
28. Wei, J., Chen, J., Feng, Y., Ferles, K., Dillig, I.: Singularity: Pattern fuzzing for worst case complexity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 213–223 (2018)
29. Wüstholz, V., Olivo, O., Heule, M.J., Dillig, I.: Static detection of dos vulnerabilities in programs that use regular expressions. In: Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II 23. pp. 3–20. Springer (2017)
30. Xia, C.S., Paltenghi, M., Le Tian, J., Pradel, M., Zhang, L.: Fuzz4all: Universal fuzzing with large language models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13 (2024)
31. Yang, C., Chen, J., Lin, B., Wang, Z., Zhou, J.: Advancing code coverage: Incorporating program analysis with large language models. ACM Transactions on Software Engineering and Methodology (2025)
32. Žikelić, Đ., Chang, B.Y.E., Bolignano, P., Raimondi, F.: Differential cost analysis with simultaneous potentials and anti-potentials. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 442–457 (2022)
33. Zlomislić, V., Fertalj, K., Sruk, V.: Denial of service attacks, defences and research challenges. Cluster Computing **20**, 661–671 (2017)