

# Can LLM Generate Regression Tests for Software Commits?

JING LIU\*, U.C. Irvine and MPI-SP, USA and Germany

SEONGMIN LEE\*, MPI-SP, Germany

ELEONORA LOSIOUK, University of Padova, Italy

MARCEL BÖHME, MPI-SP, Germany

Large Language Models (LLMs) have shown tremendous promise in automated software engineering. In this paper, we investigate the opportunities of LLMs for automatic regression test generation for programs that take highly structured, human-readable inputs, such as XML parsers or JavaScript interpreters. Concretely, we explore the following regression test generation scenarios for such programs that have so far been difficult to test automatically in the absence of corresponding input grammars:

- *Bug finding*. Given a code change (e.g., a commit or pull request), our LLM-based approach generates a test case with the objective of revealing any bugs that might be introduced if that change is applied.
- *Patch testing*. Given a patch, our LLM-based approach generates a test case that fails before but passes after the patch. This test can be added to the regression test suite to catch similar bugs in the future.

We implement CLEVEREST, a feedback-directed, zero-shot LLM-based regression test generation technique, and evaluate its effectiveness on 22 commits to three subject programs: Mujs, Libxml2, and Poppler. For programs using more human-readable file formats, like XML or JavaScript, we found CLEVEREST performed very well. It generated easy-to-understand bug-revealing or bug-reproduction test cases for the majority of commits in just under three minutes—even when only the code diff or commit message (unless it was too vague) was given. For programs with more compact file formats, like PDF, as expected, it struggled to generate effective test cases. However, the LLM-supplied test cases are not very far from becoming effective (e.g., when used as a seed by a greybox fuzzer or as a starting point by the developer).

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; *Software evolution*; *Software maintenance tools*.

Additional Key Words and Phrases: Cleverest, Regression test generation, LLM

## ACM Reference Format:

Jing Liu, Seongmin Lee, Eleonora Losiouk, and Marcel Böhme. 2024. Can LLM Generate Regression Tests for Software Commits?. 1, 1 (September 2024), 21 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 Introduction

Recently, Large Language Models (LLMs) have shown tremendous promise. Liu et al. [14] just published a comprehensive survey of research on the automation of the most important software engineering processes, including requirements engineering, code generation, program analysis, testing, debugging, and end-to-end development and maintenance. For instance, the LLM-based AutoCodeRover [36] can successfully resolve more than 30% of Github issues in a recent benchmark.

In this paper, we take a critical perspective on the utility of LLMs as assistants in automated *regression test generation for programs that take human-readable, highly structured inputs*. Given a commit or pull request, regression test generation is the problem of exposing any bugs that may have been introduced or fixed by the code changes in that commit. We find this specific problem statement interesting for several reasons.

---

\*Both authors contributed equally to this research.

---

Authors' Contact Information: Jing Liu, U.C. Irvine and MPI-SP, USA and Germany; Seongmin Lee, MPI-SP, Germany, seongmin.lee@mpi-sp.org; Eleonora Losiouk, University of Padova, Italy; Marcel Böhme, MPI-SP, Germany.

- (1) Regression test generation is an *important practical problem*. Given a code change (e.g., a commit or pull request), developers and code reviewers would like to have a witness test case<sup>1</sup> that checks (i) whether the change introduced any bugs or (ii) if a patch really fixed the bug.
- (2) Regression test generation is an *important research problem* with a large body of existing work along different streams of research, including symbolic execution, search-based software testing, and directed or regression greybox fuzzing. However, we are unaware of existing LLM-based approaches to generate bug-revealing test cases from code commits.
- (3) Our problem poses an *interesting opportunity* for LLMs. While existing regression testing techniques [10, 20, 33, 38], without existing seed inputs or input grammars, often fail to generate valid inputs in the required format for programs taking highly structured inputs, LLMs are often found to generate grammatically correct text in any language (incl. programming languages).
- (4) Our problem poses *interesting challenges* for LLMs: (a) An LLM has no access to the source code except for the commit message and diff (i.e., the changed lines of code plus a few unchanged lines for context). Without the full code, how can an LLM interpret the behavior to be tested? How does it perform with only a commit message or diff? (b) Even if an LLM “understands” the changed unit (i.e., function or feature), how can it generate a system-level input that exercises that unit and exposes a behavioral difference? (c) LLMs are known to struggle with reasoning [8]. Yet, the regression testing problem is inherently a program analysis problem that requires the capability to reason about the outcome of the computation represented by the program statements. With these challenges in mind, we conduct experiments to explore the utility of LLMs in regression test generation.

To evaluate the utility of LLMs for regression testing, we develop CLEVEREST, a feedback-directed, zero-shot LLM-based regression testing technique for programs that take highly structured, human-readable inputs. Using CLEVEREST, our evaluation seeks to answer the following research questions:

- RQ1 - How well does CLEVEREST perform as a regression test generator?
- RQ2 - How does CLEVEREST perform under various hyperparameter values?
- RQ3 - How does CLEVEREST compare to WAFLGo, the state-of-the-art in regression testing?

*RQ1. How well does CLEVEREST perform as a regression test generator?* For the XML parser and JavaScript interpreter, CLEVEREST performed unexpectedly well. In under three minutes, it found bugs in 3 of 6 bug-introducing commits and reproduced bugs patched in 4 of 6 bug-fixing commits—it at least reached the changed code in 4 of the remaining 5 cases.<sup>2</sup> However, for the PDF parser, which requires a complex input format, CLEVEREST could not find or reproduce any of the five bugs. It did reach the changed code in half the commits and exposed a difference in one.

*RQ2. How does CLEVEREST perform under various hyperparameter values?* To gain a more holistic perspective of the capabilities of CLEVEREST, we studied the impact of various hyperparameters on effectiveness. Most interestingly, we find that CLEVEREST continues to generate effective test cases even when only an expressive commit message is provided. An LLM-based approach seems to find bugs in the changed feature as long as the intention is obvious from the commit information regarding which feature has been changed. We also found that asking CLEVEREST to generate the command line prompt to execute the test case itself did *not* reduce its performance; it even found a bug that was related to the changed feature but was only fixed much later. In addition, amongst

<sup>1</sup>Throughout this paper, we use the terms “test case” and “test input” interchangeably.

<sup>2</sup>We also checked for data leakage, i.e., whether the generated test cases existed in the LLM’s training data. We computed the similarity, using Levenshtein distance ratio [2], between CLEVEREST-generated test cases and available bug-triggering test cases from the bug report (not the commit). The majority of CLEVEREST-generated cases were less than 7% similar (mean 10%, max 40%), suggesting they are generated based on commit information, not memorized from training data.

others, we found that using the less powerful GPT-4o mini or dropping the execution feedback is detrimental to its effectiveness.

*RQ3. How does CLEVEREST compare to WAFLGo, the state-of-the-art in regression test generation?* WAFLGo [33] is the most recent, state-of-the-art directed greybox fuzzer for regression test generation that was shown to outperform previous directed and regression greybox fuzzers [4, 6, 15, 37, 38]. Given a corpus of valid input seeds, WAFLGo mutates these seeds in a feedback-directed manner with the objective of reaching the changed code and exposing any bugs. In our experiments, we found that, while substantially faster (under 3 minutes versus several hours), CLEVEREST performs as well as WAFLGo in bug reproduction and slightly worse than WAFLGo in bug finding. However, upon closer inspection, we found that CLEVEREST was often not very far from exposing the bug. In the majority of cases, CLEVEREST would at least reveal a difference in behavior. Only for 5 of 22 commits would it not even reach the changed code (short of generating valid PDF files). Indeed, running a fuzzer on the CLEVEREST-generated seeds, we found/reproduced more bugs than WAFLGo when started on the independently provided initial seeds.

We also investigated the opportunities of CLEVEREST as a zero-shot regression test generator. While CLEVEREST generates the regression test cases from scratch, the performance of WAFLGo depends on an initial set of seed files (i.e., valid XML, JS, or PDF files). Upon closer inspection, we found that WAFLGo’s initial seeds happen to be very close to bug-revealing already:<sup>3</sup> Out of 22 cases, in 4 cases, the initial seeds revealed the bug; in 9 cases, the initial seeds reached the changed code, and in 2 cases, only a few characters needed to be changed to reveal the bug. In contrast, by upgrading CLEVEREST by fuzzing the generated test cases, our zero-shot approach outperforms WAFLGo, whose performance depends on a user-provided seed corpus. Hence, we find that CLEVEREST, as a zero-shot regression test generator, also makes an excellent seed generator for greybox fuzzers.

In summary, this paper makes the following contributions:

- We introduce CLEVEREST, a feedback-directed, zero-shot LLM-based regression test generation technique for programs that take highly structured, human-readable inputs to evaluate the utility of LLMs for regression test generation.
- We evaluate CLEVEREST on 22 commits to three subject programs: Mujs, Libxml2, and Poppler. We find that CLEVEREST performs very well for programs using human-readable file formats.
- CLEVEREST performs robustly even when only the code diff is given, without the command line prompt, or the randomness of the LLM is increased. Lack of information in the commit message, a less powerful LLM, or dropping the execution feedback is detrimental to its effectiveness.
- CLEVEREST performs closely while being substantially faster than WAFLGo, a state-of-the-art few-shot regression test generator, whose initial seeds are already close to bug-revealing. Using CLEVEREST-generated test cases as seeds, a vanilla AFL++ outperforms WAFLGo.
- All the implementation code, data, and scripts used in this study are shared in the replication package available at <https://anonymous.4open.science/r/cleverest-7767/>.

Overall, we can safely recommend the use of LLMs as an effective means to automatically generate or even just bootstrap the manual generation of regression test cases for a code commit or pull request—particularly if the program takes highly structured, human-readable inputs as test cases.

## 2 Experimental Design

In this paper, we seek to evaluate the capabilities of a large language model (LLM) as a regression test generation tool in the CI/CD pipeline (Continuous Integration / Continuous Delivery) for

<sup>3</sup>For fairness, the WAFLGo authors decided to take the initial seeds from the UNIFUZZ benchmark [13].

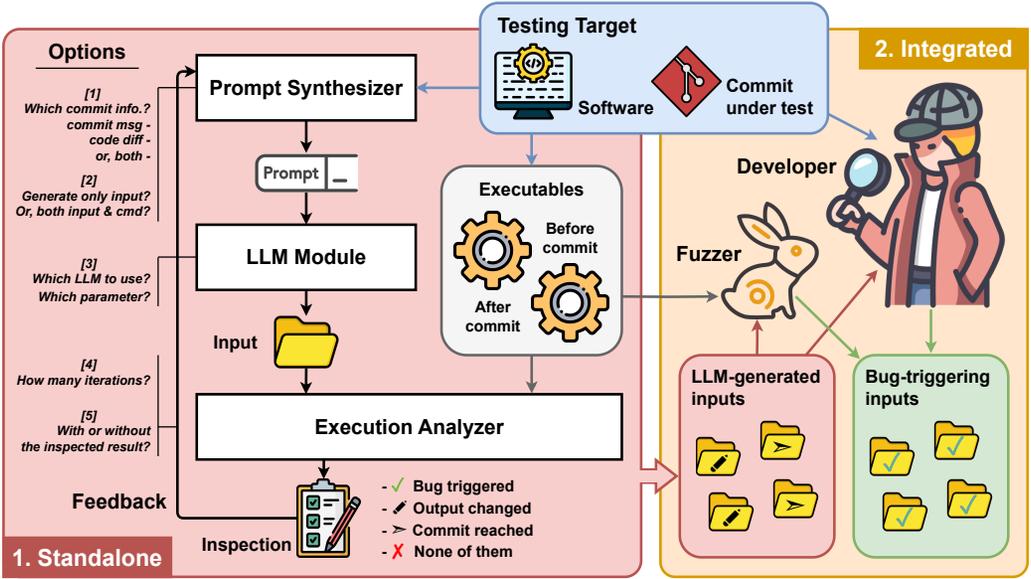


Fig. 1. CLEVEREST: Our zero-shot feedback-guided LLM-based regression test generation methodology.

programs that take highly structured, human-readable inputs, like JavaScript programs or XML files. Given only the code changes and/or a description of these changes (i.e., a commit or pull request), we want to find out how well an LLM performs in generating test cases that reveal bugs that might have been introduced or patched by these changes. Rather than using an LLM as-is, we introduce a tailored approach to our regression testing problem statement where the LLM is embedded in a feedback-directed framework involving a prompt synthesizer and execution analyzer. We call our tool CLEVEREST (communicate with an LLM for software version testing). Finally, we discuss our benchmark selection and experimental infrastructure.

## 2.1 Research Questions

- *RQ1. How well does CLEVEREST perform as a regression test generator?* Specifically, we evaluate the two most important properties of CLEVEREST if it was fully embedded in a CI/CD pipeline: effectiveness and execution time. Given the corresponding commits, we evaluate (i) finding bugs that were introduced in a commit and (ii) reproducing bugs that were patched in a commit. To understand how CLEVEREST performs in the hands of a developer, we study quantitatively and qualitatively how “close” CLEVEREST-generated test cases are from revealing the bug.
- *RQ2. How well does CLEVEREST perform under various hyperparameter values?* Specifically, we evaluate how well CLEVEREST performs compared to the default configuration if we (a) only used the commit message but not the diff, (b) only used the commit diff but not the message, (c) let it generate the command line prompt in addition to the test input. (d) used maximum LLM temperature, (e) used a less powerful LLM (4o-mini), (f) used 10 iterations instead of 5 in the feedback loop, and (g) dropped the execution analysis result in the feedback loop.
- *RQ3. How does CLEVEREST compare to WAFLGo, the state-of-the-art in regression test generation?* Specifically, we evaluate (1) the performance of WAFLGo as a few-shot regression test generator and (2) the dependence of WAFLGo on the quality of the initial seed corpus. We then (3) evaluate

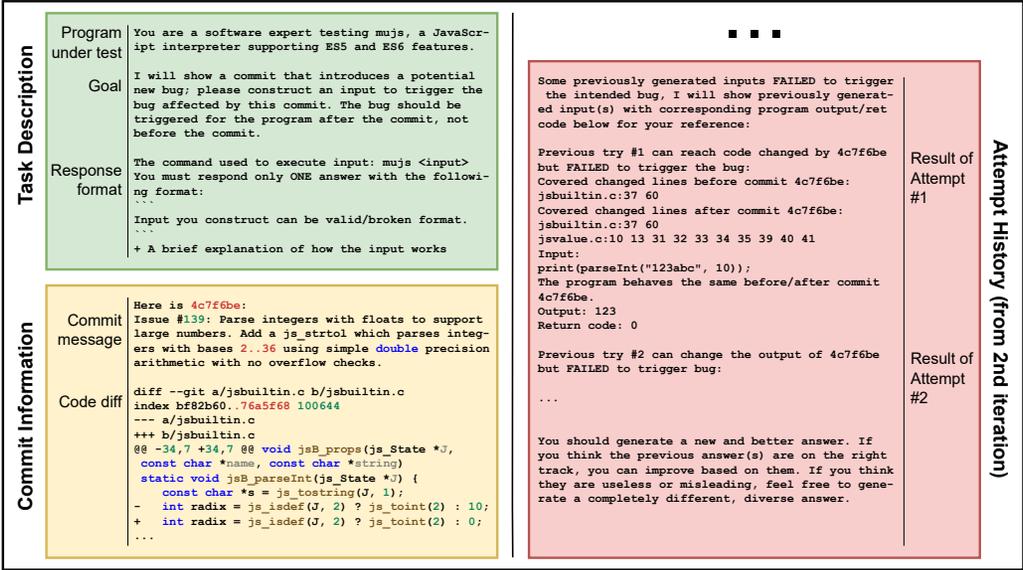


Fig. 2. The prompt generated by the prompt synthesizer for the LLM to generate the test input for Mujs.

the performance of CLEVEREST against WAFLGo both directly and as a seed generator for zero-shot greybox fuzzing.

## 2.2 CLEVEREST: LLM-based Regression Testing Technique and Implementation

We present the design and implementation of CLEVEREST, a zero-shot feedback-guided LLM-based regression test generation tool, to evaluate the effectiveness of LLMs for generating regression tests for software commits to programs taking highly structured, human-readable inputs.

Figure 1 gives a procedural overview of CLEVEREST. Our tool CLEVEREST consists of three key components: *Prompt Synthesizer*, *LLM Module*, and *Execution Analyzer*, which work in a *feedback loop* to generate regression test cases for the commit under test.

**2.2.1 Prompt Synthesizer.** The first component of CLEVEREST is the *Prompt Synthesizer*, which generates a domain-specific prompt for the LLM to create the test input. Given the testing target, which is the software and the commit under test, and optionally some feedback from a previous iteration, the prompt synthesizer constructs the prompt for the LLM, which includes the task description, commit information, and the attempt history.

Figure 2 shows an example prompt to generate a regression test case for a code commit to Mujs, a lightweight JavaScript interpreter. The synthesized prompt has the following structure:

- (1) **Task Description:** The beginning of the prompt starts with a general description of the task. The main parts consist of (a) a simple sentence to describe the program under test, e.g., “JavaScript interpreter,” (b) the goal of the LLM-generated input, e.g., “trigger a bug introduced by the commit,” and (c) format specification of LLM’s response, e.g., “return the input wrapped in a triple-backtick fenced block along with its brief explanation of the expected behavior.”
- (2) **Commit Information:** The second part contains the information about the commit under test to the LLM. We consider both the commit message and the code diff as the source of the information as they provide different levels of context about the commit: The *code diff* contains

the changed lines of code plus a few unchanged lines of context. The *commit message* offers a high-level description of the developer’s intention and the purpose of the commit.

- (3) **Attempt History:** From the second iteration on, the prompt will contain the execution result of the previous input generated by LLM. Each item for the previous input will show (a) the general execution result analyzed by the execution analyzer (we will discuss this in Section 2.2.3), (b) the program output (combining stdout and stderr), and (c) the program return code.

Several hyperparameters in the prompt synthesizer will affect the task difficulty for the LLM. One is the level of commit information provided to the LLM: we may consider providing only the code diff, only the commit message, or both. Another hyperparameter is whether to provide the command-line utility and parameters (*cmd*) expected to be used to test the commit. Unlike fuzzing tools, which can only modify the input and require the user to specify the *cmd*, an LLM can generate the full *cmd* for the program execution. Allowing the generation of the *cmd* can make the task more challenging for the LLM, as it needs to understand the program’s behavior to generate the correct *cmd*.

**2.2.2 LLM Module.** The generated prompt is fed into the LLM module, which generates the test input for the commit under test. The output of the LLM contains the content of the test input along with the explanation of the expected behavior and the *cmd* if asked. The output is then parsed to produce the actual test case, which is executed to verify the commit under test. Depending on the task description, the LLM tries to generate the input that triggers the bug introduced by the commit (without knowing a priori if the bug is introduced) or reproduces the bug fixed by the commit.

The LLM module has several hyperparameters that affect the quality of the generated output. One hyperparameter is the *size* of the LLM model, which determines the model’s capacity. A larger model size generally results in better output quality but requires more computational resources. Another hyperparameter is the *temperature*, which controls the randomness of the output. A higher temperature value results in more randomness in the output, while a lower temperature value results in more deterministic output.

**2.2.3 Execution Analyzer and Feedback Loop.** The final component of CLEVEREST, the *Execution Analyzer*, receives the LLM-generated test input and executes it on the program before and after the commit under test. The execution analyzer compares the two executions to measure if the generated input contributes to the commit testing. To classify the test outcome, we leverage the RIPR model [1, 12], i.e., Reaching, Infecting, Propagating, and Revealing, to measure the effectiveness of the generated input. We measure the result of the execution in the following three aspects:

- (1) **Bug Triggering:** Bug triggering is the ultimate goal of the testing; it represents the *Revealing* part of the RIPR model. Depending on the scenario, the execution analyzer checks if the input triggers the intended bug: the input should trigger the bug on the program after or before the commit for the bug-finding or bug-reproduction scenario, respectively, and the same input should not trigger the same bug on the other version of the program. For instance, even if the input triggers a bug in the program, it is not considered a successful bug triggering if the same bug is triggered in both versions of the program.
- (2) **Output Changing:** Even if the input does not trigger a bug as the sanitizer does not detect the anomaly, it may result in a behavior difference; this represents the *Infecting* and *Propagating* parts of the RIPR model. We detect the behavior difference by comparing the output and return code of the program before and after the commit. Any difference will imply that the generated test successfully exploits the difference in the program behavior introduced by the commit.
- (3) **Commit Reaching:** The first necessary condition for input to reveal a bug introduced by the commit is to reach the code changed by the commit; this represents the *Reaching* part of

the RIPR model. Yet, even generating an input that reaches the commit is not trivial. We thus measure the coverage of the program execution with the generated input to see if there is any overlap between the code changed by the commit and the code covered by the execution.

The output of the execution analyzer is the assessment result of the generated input categorized into four types: bug triggered, output changed, commit reached, and none of the above. Notice that the prior categories are strict inclusions of the latter ones.

If the execution results do not trigger a bug, the inspected execution result, i.e., whether the input reaches the commit, if so, which lines are covered, and if the execution result is different, is transmitted back to the prompt synthesizer as feedback for the next input generation. This incremental prompting strategy can guide the LLM in exploring the input space more effectively and being able to generate high-quality test input for the commit under test.

Two hyperparameters in the feedback loop affect the performance of CLEVEREST. One hyperparameter is whether or not to provide the analyzed execution result as feedback to the LLM for the next iteration. If not, only the previously generated input is used as the prompt for the next iteration to avoid generating the same input. The other hyperparameter is the number of iterations, which determines the number of times the LLM generates the input, and the execution analyzer verifies the input. A higher number of iterations can lead to more effective input generation but requires more computational resources.

*2.2.4 Implementation.* Below, we describe the implementation details of CLEVEREST.

*Commit Information Extraction.* We use Git to extract the commit message and code diff. The command `'git show -format=%B'` is used to print them together in a concise format. For commits with lengths that exceed the context length of LLM, we will apply a filter mechanism to keep only code diff in the C/C++ source file. If it still does not fit, we keep only the commit message.

*Execution Feedback.* CLEVEREST utilizes the sanitizer while compiling the program to detect the bugs. It inserts instrumentation code during the compilation so that it detects malicious or undefined behavior at runtime. Using the sanitizer, one can detect various types of bugs, such as memory corruption, use-after-free, and buffer overflow, without any preliminary effort to define the oracle and write assertions or test cases.

In the experiment, we used AddressSanitizer since the benchmark dataset we considered consists of memory-related bugs. The execution analyzer checks for the keyword Sanitizer in the program output to determine the existence of a bug. For behavior difference, we compare the stdout, stderr, and return code of the program built before and after the commit under test. To identify the reachability of the commit, we use GCOV to get code coverage information. We consider the input reaches the commit if there is any overlap between the code changed by the commit and the code covered by the execution.

*LLM Configuration.* We use `openai-cli`, a command line client written in Bash, to make LLM queries. We use GPT-4o as the default model and set the max token to 4096 to achieve a balance between performance and cost. It is worth noting that we do not keep conversation history when querying LLM, as it consumes more tokens. The default temperature is set to 0.5, and the number of iterations is set to 5. We additionally consider up to ten iterations and the temperature to be 1.0 to investigate the effect of the parameter in our methodology. We also consider the GPT-4o mini model to investigate the effect of the model size in our methodology. GPT-4o mini is a smaller, more efficient version of GPT-4o; it scores 82% on the MMLU benchmark, whereas GPT-4o scores 88.7%. It is designed especially for applications where affordability and lower latency are critical.

Table 1. Detail of Bug Dataset. The column **Issue** refers to the issue number in the project repository. The columns **BIC** and **BFC** refer to the bug-introducing-commit and the bug-fixing-commit id, respectively.

Software	Description	Command-Line Utility and Parameters	Issue	BIC	BFC
Mujs	JavaScript interpreter	mujs (input file)	#65	8c27b12	833f82c
		mujs (input file)	#141	832e069	6871e5b
		mujs (input file)	#145	4c7f6be	f93d245
		mujs (input file)	#166	3f71a1c	8b5ba20
Libxml2	XML parsing library	xmllint -recover -sax1 -sax (input file)	#535	9a82b94	d0c3f01
		xmllint -recover -dropdtd -nofixup-base-uris -sax1 (input file)	#550	7e3f469	6273df6
Poppler	PDF rendering library	pdfunite (input file 1) (input file 2) output.pdf	#1282	3d35d20	4564a00
		pdfunite (input file 1) (input file 2) output.pdf	#1289	3cae777	efb6868
		pdftops (input file) /dev/null	#1303	e674ca6	a4ca3a9
		pdftoppm -mono -cropbox (input file)	#1305	aaf2e80	907d05a
		pdftoppm -mono -cropbox (input file)	#1381	245abad	1be35ee

### 2.3 Benchmark Selection and Experimental Infrastructure

*Benchmark Selection.* To answer the research questions for CLEVEREST, we choose a commit benchmark dataset for our experiment based on the following *selection criteria*:

- (1) We want to focus on regression test generation for programs that take highly structured, human-readable input formats.<sup>4</sup>
- (2) We want to maximize the fairness of our comparison to the state-of-the-art regression test generation tool WAFLGo, which is used for comparison.
- (3) We consider real-world regression bugs of various types that have been introduced and patched in identified commits to open-source software.

Table 1 shows the details of the dataset that satisfies our selection criteria. Specifically, we found that the benchmark dataset that was used during the evaluation of WAFLGo [33] satisfies selection criteria 2 and 3, while the three programs (i.e., the JavaScript interpreter Mujs, the XML parsing library Libxml2, and the PDF rendering library Poppler) satisfy selection criterion 1. These programs cover various types of textual input formats: JavaScript, XML, and PDF, which are all highly structured. Each software has one or more command-line utilities that retrieve the textual input files as command-line parameters with relevant options.

The WAFLGo dataset lists 11 bugs for these three benchmark programs, all of which are memory-related. The bug types span a wide range of memory-related bugs, including heap-buffer-overflow, global-buffer-overflow, heap-use-after-free, stack-overflow, etc. For each bug, we consider the bug-introducing-commit (BIC) and the bug-fixing-commit (BFC) as the target commit in each scenario: BIC for bug-finding and BFC for bug-reproduction scenarios. Thus, we have 22 commits in total for the experiment.

*Experiment Configuration.* The default configuration of our CLEVEREST is as follows: we provide both the commit message and the code diff to the LLM as the commit information, and we provide the command-line utility and parameters expected to be used to test the commit. We attached the execution analysis result during the feedback. As previously mentioned, we use the GPT-4o model with a temperature of 0.5 and set the number of iterations to 5. We set the timeout for each LLM-based input generation to 30 seconds. We repeat each experiment five times to account for the randomness of the LLM and the fuzzer. All experiments are conducted on a docker container with 64 cores of AMD EPYC 7713P @ 2.0GHz and 251GB memory.

<sup>4</sup>Without valid input examples as seeds or an input grammar this type of programs pose a challenge for existing regression test generation tools while we LLMs are known to handle highly structured texts quite well.

Table 2. Results for bug finding and bug reproduction effectiveness of CLEVEREST across five repetitions. For every bug and both scenarios, we show the average effectiveness score on a slider (✘ : Not reached; > : reached; 📎 : output-changing; ✓ : bug-revealing, the number of trials in which the bug was found (“Bug”), and the average time in seconds (“T (s)”).

		Bug-Introducing Commit				Bug-Fixing Commit							
		✘	>	📎	✓	Bug	T (s)	✘	>	📎	✓	Bug	T (s)
Mujs	#65					5/5	14.2					2/5	26.6
	#141					0/5	47.0					0/5	33.2
	#145					4/5	31.6					3/5	29.4
	#166					0/5	182.6					5/5	12.6
Libxml2	#535					5/5	8.8					5/5	7.8
	#550					0/5	51.4					0/5	39.8
Poppler	#1282					0/5	162.2					0/5	155.0
	#1289					0/5	131.8					0/5	89.2
	#1303					0/5	188.4					0/5	123.2
	#1305					0/5	185.0					0/5	152.4
	#1381					0/5	161.0					0/5	133.6
						3/11	105.8					4/11	73.0

### 3 RQ1. Evaluation of Capabilities

*Effectiveness.* We measure the effectiveness of CLEVEREST by computing the *average effectiveness score* across all five repetitions. If the generated regression test case finds the bug in the bug-introducing commit or reproduces the bug in the bug-fixing commit, the score is 3 (✓). Similarly, 2 indicates an output-changing test case (📎), 1 indicates a commit-reaching test case (>), and 0 indicates a test case that fails to reach the changed code (✘). Visually, we represent the average effectiveness score on a slider. The slider is colored in red, orange, olive, and teal if the score is within the range of [0-0], (0-1], (1-2], and (2-3], respectively.

Table 2 shows how well CLEVEREST performs as a regression test generator. We find that CLEVEREST performs unexpectedly well in bug finding and bug reproduction for the two programs, Mujs and Libxml2, which take more human-readable formats. Despite missing the code of the full program and without information about the input features needed to exercise the changed code, CLEVEREST found bugs in 3 of 6 bug-introducing commits (one bug existed before) and reproduced the bugs patched in 4 of 6 bug-fixing commits—it at least reached the changed code in 4 of the 5 remaining cases.

However, for the Poppler PDF parser, which requires a complex input format, CLEVEREST could not find or reproduce any of the five bugs. At least, it reached the changed code in half of the commits and exposed a difference in one commit. CLEVEREST consistently fails to generate the input relevant to the regression test generation in both scenarios for the two issues of Poppler, #1305 and #1381. While the generated input contains some related components required to test the commit, it does not satisfy the validity constraints required to successfully parse the input. For instance, CLEVEREST can generate an input that already contains the key elements necessary for revealing the bug in Issue #1305, including an annotation of type Highlight with an appearance stream and a Resources dictionary with an ExtGState entry. The failure to reach the commit occurs because Poppler strictly checks the existence of QuadPoints and Rect properties in the annotation object and exits early before reaching the changed code. If CLEVEREST knew these validity constraints and added these specific elements, as we confirmed, the generated PDF input could have reached the commit and even demonstrated an output difference before and after the commit.

*Execution time.* CLEVEREST is definitely suited for the time-constrained environment of a CI/CD pipeline. CLEVEREST takes less than one minute on average for generating (XML and JS) inputs for Mujs and Libxml2 (~ 0.2\$ on OpenAI), and still less than five minutes for more (PDF) inputs which require a more complex input format (~ 0.5\$). One exception is the case of Mujs #166 in the bug-finding scenario, where the LLM generates an input with an enormous array size to trigger the bug, which usually takes a long time to execute in the Execution Analyzer.

*Utility of CLEVEREST-generated test cases.* We evaluate how developers can use the CLEVEREST-generated input in the pursuit of commit testing. Based on our execution analysis result, we found that even when CLEVEREST-generated inputs do not directly trigger the bug, the commit-reaching or output-changing inputs are still useful for the human developer to understand the program behavior and to guide the bug-finding process. CLEVEREST-generated test cases for bug-introducing commits often “prepare” the precondition for the input to test the commit. For instance, the commit introducing Issue #550 of Libxml2 refines the XML parser’s logic for checking the occurrence of the ‘<’ character in entities. CLEVEREST catches the “intention” of the commit and generates an input containing a ‘<’ character for the entity (e.g., CLEVEREST generated <!ENTITY test "<notallowed">) to reach the commit. The bug appears with the command `xmlLint -dropDTD`, which removes the Document Type Definition (DTD) section from the document, which includes the entity definitions; thus, if CLEVEREST-generated input that has an entity reference, which gets removed by the command (e.g., `<!DOCTYPE[<!ENTITY test>]<?xml:stylesheet type="text/xsl" href="test.xsl" />`), the bug would have been exposed.

CLEVEREST-generated test cases for bug-fixing commits can often be modified to trigger the bug. For instance, the commit that fixes Issue #141 of Mujs adds missing end-of-string checks in the `regex` lexer for special syntax-indicating starting characters, including `\x`, `\u`, and `\c`. CLEVEREST generates the input `var regex = /\x/;` which exactly reflects the changed feature and produces a difference in program output from ‘SyntaxError: invalid escape sequence’ to ‘SyntaxError: unterminated escape sequence,’ demonstrating a newly added error-handling sequence in the commit. To trigger the bug in the program before the bug-fixing commit, the `regex` has to have a very long string that starts with one of those special characters—information that cannot be derived from the commit information alone. Yet, the human developer can easily modify the CLEVEREST-generated input to contain a long string that reproduces the bug that was fixed by that commit.

*RQ1. Result Summary.* CLEVEREST performs unexpectedly well for the commits to programs that take more human-readable formats but struggled to generate the right structure for the more complex format. With a short execution time of minutes, CLEVEREST is well-suited to be used in the CI/CD pipeline. As they are easy to read and may already be halfway there, CLEVEREST-generated inputs can also be used as a starting point for manual regression testing, even if they do not trigger any bugs in the given commit.

#### 4 RQ2. Ablation Study

Table 3 shows how CLEVEREST performs under various hyperparameter values for the bug-finding and bug-reproduction scenarios. Specifically, we evaluate how well CLEVEREST performs compared to the default configuration if we only used the commit message but not the diff or only used the commit diff but not the message (*commit information*), let it generate the command line prompt in addition to the test input (*task difficulty*), used GPT-4o mini as a less powerful LLM or maximized the LLM temperature (*LLM module*), used ten (10) iterations instead of five (5) in the feedback loop (*number of iterations*), or dropping the execution analysis result from the feedback (*feedback utility*).

Table 3. Results for our ablation study with average effectiveness score on a slider.

Scenario	Subject	Issue	Default	Prompt Synthesizer			LLM Module		Execution Analyzer	
				Only msg	Only diff	Gen. cmd	Tempo <sub>0,x</sub>	4o-mini	Iter <sub>10</sub>	No feedback
Bug-finding	Mujs	#65								
		#141								
		#145								
		#166								
	Libxml2	#535								
		#550								
	Poppler	#1282								
		#1289								
		#1303								
		#1305								
#1381										
Average										
Bug-reproduction	Mujs	#65								
		#141								
		#145								
		#166								
	Libxml2	#535								
		#550								
	Poppler	#1282								
		#1289								
		#1303								
		#1305								
#1381										
Average										

*Commit Information.* We evaluate the impact of only using the commit message but not the diff (“only msg”) or only using the commit diff but not the message (“only diff”) and find that the results reproduce as long as the *intention* of the change is captured. In the bug-finding scenario, providing only the commit message changes the score very slightly from 1.32 to 1.35 (i.e., sometimes output-changing). In both scenarios, providing only the commit diff also only changes the score very slightly from 1.32 to 1.23 and from 1.35 and 1.44, respectively.

Only in the bug reproduction scenario, where the commit messages for bug fixes happen to be fairly brief and lack detailed information about the intended change, the effectiveness of CLEVEREST drops from 1.35 (i.e., sometimes output-changing) to 0.66 (i.e., sometimes unreachable). For example, the bug fix for Issue #145 of Mujs comes with the commit message “Fix js\_strtol.” This is too brief to understand the bug context. The patch for Issue #1284 of Poppler comes with the commit message, “topIdx can’t be negative.” Yet, it remains unclear what topIdx is, how it can be negative, and how it relates to features required in the generated test case.

In contrast, the commit diff often points to the intention of the change. For instance, the patch of Issue #141 in Mujs adds an error handling routine that checks if there is an unterminated escape sequence in some regex pattern, which goes much beyond what the commit message says: “Add missing end-of-string checks in regex lexer.”

*Task Difficulty.* We evaluate the difference in effectiveness if CLEVEREST is also asked to generate the command line prompt in addition to the regression test case (“Gen. cmd”) and find no significant impact on the performance of CLEVEREST. We can only observe a negative effect for the commit introducing Issue #535 of Libxml2, changing the average score from 1.32 to 1.20. This indicates that the LLM is even capable of generating the right command line prompt for regression testing.

Interestingly, CLEVEREST found one otherwise *undiscovered bug* when asked to also generate the command line prompt. This bug was not known at the time when the benchmark data was collected and has only recently been fixed well after the GPT-4o cut-off date (Commit #3dea98e in May 2024). Specifically, when CLEVEREST was asked to generate a test case that reproduces Issue #550 of Libxml2 from the bug-fixing commit and a corresponding command line prompt, a bug was found that was unrelated to Issue #550 but related to the *same feature* that was changed (i.e., `xpath` and `dropdtd`). This confirms our intuition that CLEVEREST generates regression test cases from the intention more than the actual code changes.

*LLM Module.* We evaluate the impact of using a less powerful LLM (GPT-4o mini) and find that model size has a significant impact on the performance of CLEVEREST. Changing the LLM from GPT-4o to GPT-4o mini reduces the effectiveness of CLEVEREST for five issues in the bug-finding scenario. For two bug-introducing commits, the generated test cases never even reach the code changes. The average score drops from 1.32 to 0.81. The same happens, but more severely, for the bug-reproduction scenario: Effectiveness reduces for eight (8) patches. For five (5), the generated test cases never even reach the code changes. The average score drops from 1.35 to 0.42. The result indicates that while GPT-4o mini is more cost-effective than GPT-4o, it is insufficient to assist the complex input generation for the commit testing.

We also evaluate an increase in temperature (i.e., the degree of confabulation). We expected that maximizing temperature would also lead to an increase in test case diversity. However, we find no significant changes in the effectiveness of CLEVEREST when the temperature is increased.

*Number of Iterations.* We evaluate the impact on effectiveness if we increase the number of feedback iterations from five (5) to ten (10) and find a small positive impact only for bug-reproduction (from 1.35 to 1.56). In every iteration, CLEVEREST appends the execution feedback of the previous iteration from the Execution Analyser to the synthesized prompt (Fig. 2). We expect an increased number of iterations may positively impact on the effectiveness.

After increasing the number of iterations, two cases (2) turned from reaching to bug-triggering. Three cases (3) turned from failing to reach to output-changing. One case (1) turned from reaching to output-changing. Probably due to randomness, one case (1) turned from reaching to failing to reach. For the bug-finding scenario, we find no significant impact on CLEVEREST's effectiveness. Overall, there may be some benefit of increasing the number of feedback iterations at the cost of execution time.

*Feedback utility.* We evaluate the impact of dropping the execution analysis result from the feedback, i.e., having only the record of previously generated inputs in the prompt, and find that it has a negative impact on the effectiveness of CLEVEREST, particularly in the bug-reproduction scenario. In the bug-finding scenario (from 1.32 to 1.26), the decrease is obvious only for Issue #1289 of Poppler, where the generated test case now even fails to reach the code changes. In the bug-reproduction scenario (from 1.35 to 0.99), CLEVEREST's effectiveness reduces across all bug-fixing commits. The result indicates that the execution feedback is crucial for generating the regression test input in the bug-reproduction scenario.

Taking a closer look, we found that the feedback was especially helpful to the LLM by pointing out invalid components in the input. For instance, for Issue #166 of Mujs, the test case that is often generated in the first iteration contains a token, `Symbol`, which is not supported by the program and returns an error: `ReferenceError: 'Symbol' is not defined`. The execution feedback helps the LLM avoid generating another input with the invalid component in the next iteration – until, finally, a valid, bug-triggering input is generated.

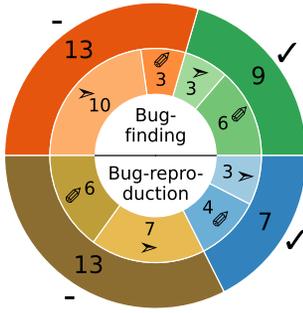
Table 4. Results for bug finding and bug reproduction effectiveness of CLEVEREST, CLEVEREST + fuzzing (CLEVFUZZ), and WAFLGo across five repetitions. For every bug and both scenarios, we show the number of initial seeds (WAFLGo #Init), the effectiveness of the initial seeds (from ✗ to ✓), the number of WAFLGo campaigns in which the bug was found and the time-to-exposure (T.O. means timeout after 24 hours), CLEVEREST’s average effectiveness score on a slider, number of bug-triggering repetitions, and execution time.

Scenario	Subject	Issue	WAFLGo			CLEVEREST			CLEVFUZZ			
			Init	Bug	T (h:m:s)	✗ >  ✓	Bug	T (h:m:s)	Bug	T (h:m:s)	Bug <sub>all</sub>	
Bug-finding	Mujs (19)	#65	✗	5/5	01:36:20		5/5	00:00:14	-/-	-	5/5	
		#141	✗	0/5	T.O.		0/5	00:00:47	4/5	09:42:15	4/5	
		#145	✗	5/5	00:14:01		4/5	00:00:31	1/1	00:00:01	5/5	
		#166	✓	5/5	00:00:00		0/5	00:03:02	0/5	T.O.	0/5	
	Libxml2 (14)	#535	✓	5/5	00:00:00		5/5	00:00:08	-/-	-	5/5	
		#550	>	0/5	T.O.		0/5	00:00:51	0/5	T.O.	0/5	
	Poppler (100)	#1282	>	5/5	00:06:23		0/5	00:02:42	4/4	00:00:55	4/5	
		#1289	>	0/5	T.O.		0/5	00:02:11	0/2	T.O.	0/5	
		#1303	✗	0/5	T.O.		0/5	00:03:08	-/-	-	0/5	
		#1305	✗	0/5	T.O.		0/5	00:03:05	-/-	-	0/5	
		#1381	>	0/5	T.O.		0/5	00:02:41	-/-	-	0/5	
	Aggregate			5/11	13:16:04		3/11	00:01:45			5/11	
	Bug-reproduction	Mujs (19)	#65	>	0/5	T.O.		2/5	00:00:26	2/3	08:11:04	4/5
			#141	✗	5/5	00:12:47		0/5	00:00:33	4/4	03:03:18	4/5
#145			✗	5/5	00:05:56		3/5	00:00:29	1/1	00:00:18	4/5	
#166			✓	5/5	00:00:00		5/5	00:00:12	-/-	-	5/5	
Libxml2 (14)		#535	✓	5/5	00:00:00		5/5	00:00:07	-/-	-	5/5	
		#550	✗	0/5	T.O.		0/5	00:00:39	-/-	-	0/5	
Poppler (100)		#1282	>	0/5	T.O.		0/5	00:02:35	0/5	T.O.	0/5	
		#1289	>	0/5	T.O.		0/5	00:01:29	0/5	T.O.	0/5	
		#1303	✗	0/5	T.O.		0/5	00:02:03	0/2	T.O.	0/5	
		#1305	>	0/5	T.O.		0/5	00:02:32	-/-	-	0/5	
		#1381	>	0/5	T.O.		0/5	00:02:13	-/-	-	0/5	
Aggregate			4/11	15:18:03		4/11	00:01:13			5/11		

*RQ2. Result Summary.* Surprisingly, we found that CLEVEREST continues to generate effective test cases that find bugs in the changed feature as long as the intention is obvious which feature is changed, i.e., even when only an expressive commit message is provided. We also found that asking CLEVEREST to generate the command line prompt itself did not reduce its performance; it even found a bug that was related to the changed feature but was only fixed much later. Reducing the model size or dropping the execution feedback both reduced the effectiveness. Doubling the number of iterations (at double the cost) slightly increased the effectiveness of CLEVEREST.

## 5 RQ3. Comparison to the State-of-the-Art

We evaluate how CLEVEREST compares to WAFLGo, the state-of-the-art in regression test generation. WAFLGo [33] is the most recent, state-of-the-art directed greybox fuzzer for regression test generation that was shown to outperform previous directed and regression greybox fuzzers [4, 6, 15, 37, 38].



(a)

Subject	Issue	Bug-finding			Bug-reproduction		
		WAFGo	CLEVEREST	CLEVFUZZ	WAFGo	CLEVEREST	CLEVFUZZ
Mujs	#65	5/5	5/5	5/5	0/5	2/5	4/5
	#141	0/5	0/5	4/5	5/5	0/5	4/5
	#145	5/5	4/5	5/5	5/5	3/5	4/5
	#166	5/5	0/5	0/5	5/5	5/5	5/5
Libxml2	#535	5/5	5/5	5/5	5/5	5/5	5/5
	#550	0/5	0/5	0/5	0/5	0/5	0/5
Poppler	#1282	5/5	0/5	4/5	0/5	0/5	0/5
	#1289	0/5	0/5	0/5	0/5	0/5	0/5
	#1303	0/5	0/5	0/5	0/5	0/5	0/5
	#1305	0/5	0/5	0/5	0/5	0/5	0/5
	#1381	0/5	0/5	0/5	0/5	0/5	0/5
Aggregate		5/11	3/11	5/11	4/11	4/11	5/11

(b)

Fig. 3. The result of the CLEVFUZZ. (a) The outer circle shows the number of cases where the fuzzer finds the bug-triggering input for the regression test generation scenario: ✓ indicates the fuzzer finds the bug-triggering input and - indicates otherwise. The inner circle shows the original result the CLEVEREST-generated input: > indicates the input reaches the commit and key indicates the input changes the program output execution. (b) The overall bug-triggering input generation result between WAFGo, CLEVEREST, and CLEVFUZZ.

Given a corpus of valid input seeds, WAFGo mutates these seeds in a feedback-directed manner, intending to reach the changed code and expose any bugs.

## 5.1 Direct Comparison

Table 4 contains the results for comparing CLEVEREST and WAFGo regarding effectiveness score and execution time. We first evaluate both tools head to head and then explore the effectiveness of CLEVEREST if the reaching or output-changing test cases were also fuzzed (CLEVFUZZ).

*Effectiveness.* Technically speaking, CLEVEREST performs as well as WAFGo in bug reproduction and slightly worse than WAFGo in bug finding (Columns *Bug* in Tab. 4). WAFGo finds the bug-triggering input in five (5) and four (4) of the 11 bug-introducing and bug-fixing commits, respectively. However, we also notice that the initial seed corpus that is provided to WAFGo already finds the bugs that were introduced and patched in the respective commits in four (4) cases (Column *Init*). We explore the dependence of WAFGo’s effectiveness further in the next section.

*Execution time.* CLEVEREST is substantially faster than WAFGo, which makes our LLM-based approach more suitable as part of the CI/CD pipeline, which runs under strict time and resource constraints. A typical fuzzing campaign is set to 24 hours. Apart from four cases where the initial seeds already found or reproduced the bug that was introduced or patched in a commit, WAFGo takes between five and fifteen minutes to find the bugs, on average. For the commit introducing Issue #65 of Mujs, WAFGo took more than 1.5 hours. CLEVEREST’s execution time is bounded by the number of iterations (fixed to 5 in our experiments) and requires less than one minute for Mujs and Libxml2 and between two and three minutes for Poppler.

*Fuzzing CLEVEREST’s seeds for improved effectiveness.* We explore an opportunity to automatically improve the effectiveness of CLEVEREST using the generated test inputs as fuzzer seeds (CLEVFUZZ). During the qualitative analysis of the change-reaching and output-changing CLEVEREST-generated test cases for RQ1 (§3), we found that they are often not “very far” from bug-revealing. This insight

is further inspired by the use of LLMs as seed generators for fuzzing as an effective strategy in the 2024 DARPA AI Cyberchallenge (AixCC).<sup>5</sup>

For every trial with a change-reaching or output-changing CLEVEREST-generate input, we run one campaign of AFL++ v4.21c with the default configuration for 24 hours using that test case as the only seed. Column CLEVFUZZ.Bug in Table 4 shows the number of successful campaigns with respect to the number of CLEVEREST trials that apply. Specifically, 22 CLEVEREST-generated test cases at least reached the changes but did not reveal the bug when trying to find the bug in six bug-introducing commits, and 20 CLEVEREST-generated test cases at least reached the change but did not reveal the bug when trying to reproduce the bug in six bug-fixing commits.

*Results.* As shown in the last column of Table 4 and Figure 3, this approach outperforms WAFLGo, which is started on a user-provided seed corpus. Specifically, our LLM-seeded fuzzer CLEVFUZZ now also finds the bugs introduced in the commits corresponding to Issue #141 of Mujs and Issue #1282 of Poppler as well as the bug that was fixed in the commit corresponding to Issue #141 of Mujs. As we can see in Figure 3.a, fuzzing turned 6 reaching and 10 output-changing CLEVEREST-generated test cases into effective test cases. This progression is further explored in Figure 3.b. Timewise, this LLM-based zero-shot fuzzing approach is still faster than WAFLGo. Considering CLEVFUZZ’s time as the sum of the time for CLEVEREST and the time for fuzzing, CLEVFUZZ takes roughly 6 hours<sup>6</sup> for both bug finding and bug reproduction scenarios, while WAFLGo takes 13 and 15 hours, respectively. Indeed, this is an intriguing result as WAFLGo can be considered few-shot (i.e., starting from a user-provided corpus) while our approach is zero-shot (i.e., no examples needed).

## 5.2 CLEVEREST as Zero-Shot Regression Test Generator

From the perspective of CLEVEREST as a zero-shot regression test generator, we wanted to explore the dependence of WAFLGo’s effectiveness on the initial set of seed files (i.e., valid XML, JS, or PDF files). For Mujs, Libxml2, and Poppler, there are 19, 14, and 100 initial seeds for WAFLGo. The authors of WAFLGo [33] used a sound and fair seed corpus selection strategy for their experiments; the initial seeds were taken from the UNIFUZZ benchmark [13]. Nevertheless, in four cases, the initial corpus already reveals Issues #166 of Mujs and #535 of Libxml2 introduced or fixed in the corresponding commits (cf. Column *WAFLGo.Init* in Tab. 4). This raises the question of how close to triggering the corresponding bugs are WAFLGo initial seeds.

```
(function() {
var a = 1, b = 2;
for(var i = 0; i < 1e5; i++) { /a/g
- if(a === b) {
+ if(a =Error== b) {
  throw new Error;
}
})();
```

Listing 1. Mujs #65

```
c = 30000;
a = [];
for (i = 0; i < 2 * c; i += 1) {
  a.push(i%c);
}
a.sort(function (x, y) { return x - y; });
- print(a[2 * c - 2]);
+ print(a[2 * a - 2]);
```

Listing 2. Mujs #145

*Results.* Table 4 (Col. *WAFLGo.Init*) shows the effectiveness of the initial corpus in terms of reaching the code changes (➤) and revealing changes in the output (👉). Among the 22 cases, in addition to the four (4) bug-revealing cases, in nine (9) cases, the initial seed corpus at least already reaches the code changes. In two (2) additional cases, only a few characters need to be changed to

<sup>5</sup>Trail of Bits, one of the finalists says: “Our system uses large language models (LLMs) to generate seed inputs for fuzzing, significantly reducing the time needed to discover vulnerabilities. This innovative approach helps us work within the competition’s strict time constraints.” <https://blog.trailofbits.com/2024/08/09/trail-of-bits-buttercup-heads-to-darpas-aixcc/>

<sup>6</sup>Including the time-out of the fuzzing campaign, which is 24 hours.

reveal the bug. Listings 1 and 2 above show such examples for Issues #65 and #145 of Mujs. Given this result, we find that CLEVEREST, as a zero-shot regression test generator, also makes an excellent seed generator for regression greybox fuzzers, like WAFLGo.

*RQ3. Result Summary.* While CLEVEREST is substantially faster than WAFLGo, CLEVEREST performs as well as WAFLGo in bug reproduction and slightly worse in bug finding. However, by upgrading CLEVEREST by fuzzing the generated test cases, our zero-shot approach outperforms WAFLGo, whose performance depends on a user-provided seed corpus, which, as we found, happens to be quite close to bug-revealing already. Hence, we find that CLEVEREST, as a zero-shot regression test generator, also makes an excellent seed generator for regression greybox fuzzers.

## 6 Threats to Validity

*Construct Validity.* A key concern is the potential for data leakage and memorization by LLMs, where test set information may inadvertently influence training. In this study, the risk pertains to LLMs memorizing the regression test cases from the WAFLGo benchmark (cf. Table 1). To assess this risk, we computed the similarity, in terms of Levenshtein ratio [2], between the CLEVEREST-generated test cases and the available bug-triggering test cases that were provided along with the bug report (not the commit). We find that the majority of CLEVEREST-generated test cases are less than 7% similar (mean 10%, max. 40%) to the ones available online. A significant difference suggests that the LLM did not simply memorize them. We also note that in an experiment where CLEVEREST was asked to generate the command line prompt, as well, a bug was found that—while unrelated to the feature changed in the targeted bug fix—was only fixed after LLM’s cut-off date (cf. RQ2).

*Internal Validity.* We identified three main threats to internal validity: implementation correctness, confabulation of the LLM, and the randomness of our results. To ensure correctness, we conducted thorough code reviews and testing. Our implementation, including data and scripts, is made available in a replication package for transparency. We used publicly accessible GPT-4o and GPT-4o mini models via the OpenAI API, adhering strictly to their documented usage guidelines. We mitigate the impact of confabulation; all generated test cases are actually executed on subject programs to validate the test outcome. To handle the randomness in our results, we repeated each experiment within the available budget, i.e., five (5) times, and reported the findings across all trials. To assess the impact of various hyperparameters on our approach effectiveness, we performed an ablation study (RQ2).

*External Validity.* We do not claim the generality of our results but consider our experiments as an important case study for three open-source C programs that take highly structured, human-readable inputs. We sought to test the capabilities of an LLM as a regression test generation tool and carefully established benchmark selection criteria to align with this goal. We selected *all* programs from the WAFLGo benchmark that apply and *all* of their commits. The results demonstrate that the LLMs are capable of generating contextually relevant and well-structured inputs for the selected commits. However, the findings may not extend to programs with less structured or non-human-readable input formats. Our case provides valuable insights into the potential of LLMs for regression test generation, expanding the current understanding of their capabilities and limitations in this context.

## 7 Related Work

*LLMs for Automatic Unit Test Generation.* The field of automated test case generation has evolved significantly, particularly with the advent of LLMs and deep learning techniques. Early research relied on non-LLM approaches, such as Atlas [31], which uses neural machine translation to

generate assert statements, and ATHENATEST [28], which applies deep learning to generate unit tests based on real-world examples. Both approaches use sequence-to-sequence models to map test methods to assert statements. Similarly, CodeT [5] relies on language models to generate both code solutions and corresponding tests for programming problems.

More recent advancements have explored LLM-based methods. For instance, Bareiß et al. [3] employ few-shot learning with large pre-trained models for code generation tasks, including unit test generation. Approaches like LLM4VV [18] and ITDCG [11] incorporate human feedback to refine the unit test generation process: LLM4VV focuses on generating unit tests for OpenACC compiler implementations, while ITDCG combines LLMs with runtime feedback for test pruning and mutation. Among the LLM-based approaches, TESTPILOT [24], COVERUP [21], HITS [30], and SymPrompt [23] explored the adoption of LLMs for automatic unit test generation. TESTPILOT [24] relies on LLMs to generate unit tests by providing the signature of the function under test, its documentation comment, its usage examples and its source code. Moreover, it sets up a feedback loop with the LLM sharing the failing test and the error message if the test fails until it gets a successful test. COVERUP [21] improves code coverage by using LLMs to generate unit tests that target untested code areas. HITS [30] breaks down complex methods into smaller slices, statically retrieves information on the dependencies of the method-to-test, and then shares it with LLM to generate high-coverage test suites. SymPrompt [23] uses LLMs to generate test inputs for executing a specific path identified during the static analysis. The LIBRO framework proposed by Kang et al. [9] is the closest solution to CLEVEREST that we found. LIBRO relies on LLMs to automatically generate test cases that reproduce bugs from general bug reports and, after ranking them, present the most relevant ones to developers.

CLEVEREST is distinguished from previous works in the following ways:

- It is the first LLMs performance evaluation with respect to regression test generation. In particular, CLEVEREST focuses on generating bug-revealing test cases from code commits, while previous works have addressed the broader unit test generation problem;
- It prompts LLMs with minimal contextual information, demonstrating that an expressive commit message is already sufficient for an LLM to generate effective test cases. On the contrary, previous works share lots of information about the target method (e.g., API signature, documentation, full source code);
- It confirms the lack of need to rely on few-shot prompting, as zero-shot prompts are already informative enough, thus not requiring samples to be shared with the LLMs;
- It does not require any prior static analysis of the target method, as the commit details (i.e., commit diff and commit message) are enough. Our feedback mechanism only calculates the code coverage and shares it with the LLM during the iterative process.

*LLMs for Static Bug Detection.* The advent of LLMs has pushed researchers towards exploring their potential in vulnerability detection. Previous works [7, 26] tested different prompting strategies [19, 27], revealing that performance varies greatly depending on the prompt, with the Chain-of-Thought approach being the most promising one. Zhang et al. [34] incorporated additional information from the source code, such as API call sequences, but found that improvements were limited to specific programming languages. To be fully integrated into developer workflows, LLMs should accurately explain the root causes of vulnerabilities. However, Ullah et al. [29] found that they often provide inaccurate explanations, frequently missing the true root causes when asked to justify their classifications. While previous works have identified limitations in LLMs when asked to identify vulnerabilities in software, in our evaluation, we saw that LLMs can perform well in generating regression test cases from code commit or pull requests, confirming they have some reasoning capabilities.

*LLMs for Fuzzing.* Several fuzzers leverage LLMs to enhance fuzzing techniques, but each operates with different goals and methods. ChatAFL [17] uses LLMs to construct grammars for protocol messages, mutate inputs, and generate sequences to improve fuzzing efficiency, distinguishing it from CLEVEREST, which is not a protocol fuzzer. PromptFuzz [16], on the other hand, utilizes LLMs to iteratively generate fuzz drivers that explore API functions, making it fundamentally different from CLEVEREST, which focuses on fuzzing through input generation rather than fuzz driver creation. Fuzz4All [32] employs a pure-LLM approach to generate and mutate code snippets testing various compiler features with user-provided documentation and example code. In contrast, CLEVEREST generates regression tests for commits in a zero-shot manner. LLM4FUZZ [25] guides fuzzing by prioritizing high-value code regions and input sequences likely to trigger vulnerabilities, serving as both a fuzzing guide and a fuzzer itself, unlike CLEVEREST. LLAMAFUZZ [35] integrates LLMs to perform structure-aware mutations, starting with traditional seeds and using the LLM to mutate them, whereas CLEVEREST uses LLMs to generate the initial seeds. Lastly, ChatFuzz [22] combines LLMs and reinforcement learning to generate interconnected machine code sequences for hardware fuzzing, a tool specifically designed to detect vulnerabilities in processors, making it quite different from CLEVEREST, which is not focused on hardware fuzzing.

## 8 Discussion

Throughout the study, we have shown the significant potential of LLMs in generating structured, human-readable, system-level inputs for regression testing. Our results highlight that even without advanced techniques such as fine-tuning or retrieval-augmented generation (RAG), LLMs, when combined with prompting and execution feedback, are capable of producing high-quality test inputs. These inputs can reflect the semantic meaning inside commits. In some cases, they directly trigger the bugs, and in other cases, they could be “repaired” to trigger the bugs. This points to the potential of LLMs in directed software testing, enabling automated testing workflows that are typically more challenging for conventional tools.

One notable observation is the LLM’s ability to generate meaningful input without access to the complete program context or compilation databases, which static analysis tools traditionally rely on. This process mirrors human intuition in crafting tests based on high-level semantic understanding. While LLMs may generate inaccurate tests, these inaccuracies are mitigated by real program execution—similar to how dynamic analysis and greybox fuzzing operate. This testing loop can be viewed as a “*clever*” fuzzer that, while slower (in terms of execs/sec) and more computationally expensive than greybox fuzzers, compensates by generating semantically rich inputs.

While using our tool CLEVEREST standalone has already shown promising results in generating effective test cases in a short time, we believe the value of CLEVEREST shines even more in their subsequent use as a means to an end (i.e., as a tool in a developer’s hand). The LLM-generated test cases are easily comprehensible by human developers, as they are structured and human-readable; we demonstrated how human developers can modify these test cases to trigger bugs. They are also an excellent seed generator for regression greybox fuzzers; even a vanilla greybox fuzzer can perform similarly or better than WAFGo, which requires a user-provided seed corpus.

This research suggests that LLMs offer a complementary tool in the evolving landscape of automated software testing. While this study focuses on text-based programs, there is an opportunity to extend CLEVEREST’s capabilities to more complex formats, such as binary files, and to environments where programs are not executed via command-line but through API calls or other non-trivial interfaces. With ongoing advancements in LLMs and further integration with execution feedback mechanisms, we anticipate even greater contributions to the field of regression testing and bug detection.

## 9 Data Availability

All the implementation code, data, and scripts used in this study are shared in the replication package available at <https://anonymous.4open.science/r/cleverest-7767/>.

## References

- [1] P. Ammann and J. Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press. <https://books.google.de/books?id=58LeDQAAQBAJ>
- [2] Max Bachmann. 2024. Levenshtein.ratio. <https://rapidfuzz.github.io/Levenshtein/levenshtein.html#ratio>.
- [3] Patrick Bareiß, Beatriz Souza, Marcelo d’Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. arXiv:2206.01335 [cs.SE] <https://arxiv.org/abs/2206.01335>
- [4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) (CCS ’17). Association for Computing Machinery, New York, NY, USA, 2329–2344. <https://doi.org/10.1145/3133956.3134020>
- [5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. CodeT: Code Generation with Generated Tests. arXiv:2207.10397 [cs.CL] <https://arxiv.org/abs/2207.10397>
- [6] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) (ICSE ’22). Association for Computing Machinery, New York, NY, USA, 2440–2451. <https://doi.org/10.1145/3510003.3510197>
- [7] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? arXiv:2310.09810 [cs.SE] <https://arxiv.org/abs/2310.09810>
- [8] Subbarao Kambhampati. 2024. Can large language models reason and plan? *Annals of the New York Academy of Sciences* 1534, 1 (2024), 15–18. <https://doi.org/10.1111/nyas.15125>
- [9] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2312–2323. <https://doi.org/10.1109/ICSE48619.2023.00194>
- [10] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. 2018. Shadow Symbolic Execution for Testing Software Patches. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 10 (sep 2018), 32 pages. <https://doi.org/10.1145/3208952>
- [11] Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2023. Interactive Code Generation via Test-Driven User-Intent Formalization. arXiv:2208.05950 [cs.SE] <https://arxiv.org/abs/2208.05950>
- [12] Nan Li and Jeff Offutt. 2017. Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering* 43, 4 (2017), 372–395. <https://doi.org/10.1109/TSE.2016.2597136>
- [13] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2777–2794. <https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei>
- [14] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large Language Model-Based Agents for Software Engineering: A Survey. arXiv:2409.02977 [cs.SE] <https://arxiv.org/abs/2409.02977>
- [15] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2693–2707. <https://doi.org/10.1109/SP46215.2023.10179296>
- [16] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation. arXiv:2312.17677 [cs.CR]
- [17] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.
- [18] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. 2024. LLM4VV: Developing LLM-driven test suite for compiler validation. *Future Generation Computer Systems* 160 (2024), 1–13. <https://doi.org/10.1016/j.future.2024.05.034>
- [19] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-Thought Prompting of Large Language Models for Discovering and Fixing Software Vulnerabilities. arXiv:2402.17230 [cs.CR] <https://arxiv.org/abs/2402.17230>
- [20] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) (PLDI ’11). Association for Computing Machinery, New York, NY, USA, 504–515. <https://doi.org/10.1145/1993498.1993558>
- [21] Juan Altmayer Pizzorno and Emery D. Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. arXiv:2403.16218 [cs.SE] <https://arxiv.org/abs/2403.16218>
- [22] Mohamadreza Rostami, Marco Chilese, Shaza Zeitouni, Rahul Kande, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. 2024. Beyond Random Inputs: A Novel ML-Based Hardware Fuzzing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. <https://doi.org/10.23919/DATE58400.2024.10546625>
- [23] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM.

- Proc. ACM Softw. Eng.*, 1, FSE (2024), 951–971. <https://doi.org/10.1145/3643769>
- [24] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. <https://doi.org/10.1109/TSE.2023.3334955>
- [25] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models. arXiv:2401.11108 [cs.CR] <https://arxiv.org/abs/2401.11108>
- [26] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. 2024. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection. arXiv:2403.17218 [cs.SE] <https://arxiv.org/abs/2403.17218>
- [27] Karl Tamberg and Hayretin Bahsi. 2024. Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study. arXiv:2405.15614 [cs.CR] <https://arxiv.org/abs/2405.15614>
- [28] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2021. Unit Test Case Generation with Transformers and Focal Context. arXiv:2009.05617 [cs.SE] <https://arxiv.org/abs/2009.05617>
- [29] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini. 2024. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 199–199. <https://doi.org/10.1109/SP54263.2024.00210>
- [30] Zejun Wang, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. <https://arxiv.org/pdf/2408.11324v1>
- [31] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM. <https://doi.org/10.1145/3377811.3380429>
- [32] Chun Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Fuzz4ALL: Universal Fuzzing with Large Language Models. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE) (2023)*, 1547–1559. <https://api.semanticscholar.org/CorpusID:260735598>
- [33] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical Code Guided Directed Greybox Fuzzing for Commits. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 2459–2474. <https://www.usenix.org/conference/usenixsecurity24/presentation/xiang-yi>
- [34] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. arXiv:2308.12697 [cs.SE] <https://arxiv.org/abs/2308.12697>
- [35] Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen. 2024. LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing. arXiv:2406.07714 [cs.CR] <https://arxiv.org/abs/2406.07714>
- [36] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE]
- [37] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1343–1360. <https://www.usenix.org/conference/usenixsecurity23/presentation/zheng>
- [38] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, Republic of Korea) (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2169–2182. <https://doi.org/10.1145/3460120.3484596>