

000 AUTOMATED BENCHMARK GENERATION FOR 001 002 REPOSITORY-LEVEL CODE INPUT SYNTHESIS VIA 003 004 COVERAGE-GUIDED FUZZING

005
006 **Anonymous authors**
007 Paper under double-blind review
008
009
010
011
012

ABSTRACT

013 Evaluating the capabilities of large language models (LLMs) on practical,
014 repository-level testing tasks is crucial for their effective application in software
015 engineering. Many existing benchmarks rely on human-authored data such as
016 issues, patches, and unit tests, which can limit scalability and introduce risks
017 of solution leakage from training corpora. We introduce TTG-GEN, an auto-
018 mated framework for generating targeted test-input generation (TTG) problems
019 from real-world codebases, in which LLMs are tasked with synthesizing input
020 byte sequences to execute specific, designated code locations. These problems
021 are representative of tasks performed by software engineers during debugging and
022 are designed to probe an LLM’s understanding of complex control and data flow
023 in real-world scenarios. TTG-GEN leverages coverage-guided fuzzing (CGF) to
024 identify reachable yet non-trivial target locations that require structure-aware in-
025 puts to cover. By automatically generating TTG problems, TTG-GEN offers a
026 practical, scalable, and continuously updatable framework with a low risk of di-
027 rect solution leakage, suited for evaluating repository-level code comprehension.
028 Using TTG-GEN, we construct TTG-BENCH-LITE, a benchmark of 500 such
029 problems derived from 16 foundational C/C++ software projects. Our evaluation
030 across retrieval-based and agent-based settings shows that even the most capable
031 LLMs solve only 15% of these problems on their first attempt. This indicates
032 that comprehending and manipulating program behavior at the repository level
033 remains a significant hurdle for current models, highlighting a substantial gap be-
034 tween their current abilities and the proficiency required for complex software
035 engineering tasks.
036
037

1 INTRODUCTION

038 In recent years, large language models (LLMs) have demonstrated significant improvements in code
039 synthesis, completion, and resolving real-world software issues (Jiang et al., 2024; Zhang et al.,
040 2023), making their application across the entire software engineering (SE) lifecycle, including de-
041 velopment, testing, and maintenance, highly promising (Hou et al., 2023; Jin et al., 2024). Moreover,
042 SE tasks often require models to understand and coordinate changes across multiple files and pro-
043 cess extensive contexts, providing a rich and sustainable testbed for evaluating the capabilities of
044 LLMs (Hudson et al., 2024; Koohestani et al., 2025). Therefore, the evaluation of LLMs on realistic
045 SE tasks is crucial for both understanding their current abilities and effectively applying them to
046 practical software tasks.
047

048 Many benchmarks have been proposed to evaluate LLM capabilities on complex SE tasks, ranging
049 from issue resolution to unit test generation (Gu et al., 2024; Jimenez et al., 2023; Mündler et al.,
050 2024; Xu et al., 2025b). However, these benchmarks exhibit certain limitations: they are either (1)
051 unrepresentative of real-world scenarios, or (2) rely heavily on human-written issues, patches, or
052 unit tests for problem creation and result verification. Consequently, this reliance limits scalability,
053 can lead to benchmark saturation, and introduces a risk of models recalling solutions seen during
pre-training (Hudson et al., 2024; Koohestani et al., 2025; Cheng et al., 2025; Dong et al., 2024).
Therefore, there is an urgent need for evaluation frameworks that are programmatically generated

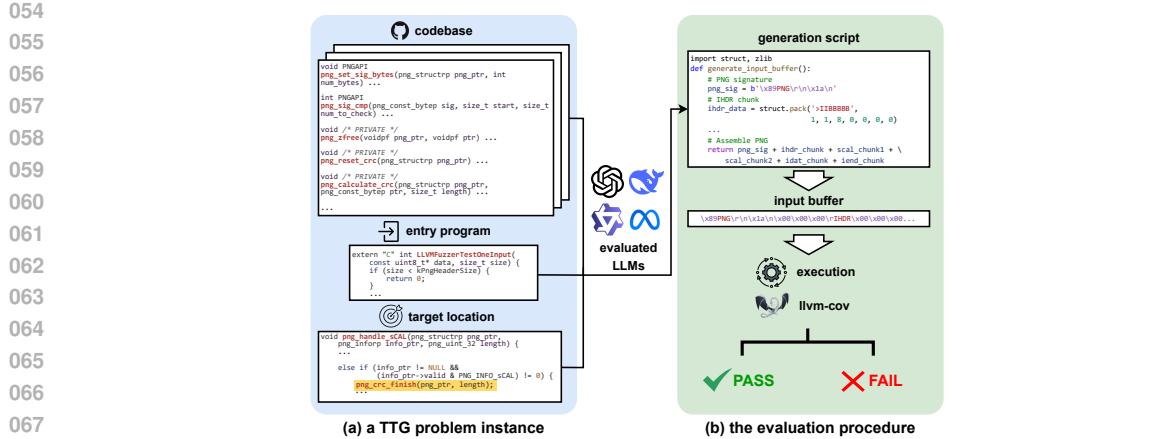


Figure 1: The TTG problem and its evaluation process. (a) A TTG problem instance, comprising a codebase, an entry program, and a target code location. (b) The evaluation procedure, where the LLM-synthesized input is executed and the coverage is analyzed to determine if the target is reached.

for scalability, carry a low risk of solution leakage, and are grounded in realistic software testing scenarios.

The targeted test-input generation (TTG) problem (see Figure 1(a) for an instance), an important task in software testing that involves creating inputs to execute specific code locations (Böhme et al., 2017; Wang et al., 2020), is well-suited for evaluating LLMs. As a benchmark, TTG allows for scalable problem generation with a low risk of solution leakage, and success is unambiguously verified through execution (Pezzè & Young, 2008; Clarke et al., 2018). A key difficulty, however, is selecting suitable targets, since random sampling yields many unreachable or trivial locations that do not adequately test a model’s capabilities (Babić et al., 2019; Metzman et al., 2021; Hamlet, 1994; Arcuri et al., 2011). To overcome this, we leverage coverage-guided fuzzing (CGF) to identify locations that are demonstrably reachable yet non-trivial to cover (Zhu et al., 2022; Manès et al., 2019; Li et al., 2018; Mallissery & Wu, 2023). Our key insight is that targets discovered in later CGF stages require more structure-aware inputs. We present TTG-GEN, a method that uses CGF and llvm-cov (LLVM, 2025) to systematically select these newly covered locations, ensuring the generation of high-quality problems for assessing system-level code comprehension (Section 3.1.3).

We constructed TTG-BENCH-LITE, a benchmark designed to evaluate LLMs’ code reasoning abilities in real-world scenarios using TTG problems. For this purpose, 16 real-world open-source C/C++ projects from a wide range of domains were selected as inputs to TTG-GEN. From the resulting set of target locations, 500 were randomly selected to form the benchmark. We evaluated several commonly used LLMs on TTG-BENCH-LITE. To handle the large scale of the code repositories, two standard settings (retrieval-based and agent-based) were employed. The result shows that the best-performing model achieved a pass@1 rate of 14.56%, highlighting the challenge presented by TTG-GEN to LLMs. In some cases, LLMs performed similarly to human experts, but in others, they made mistakes, showing their deficiencies in precise reasoning, plausible reasoning and knowledge application, indicating that there is a large room for improvement in their reasoning capabilities. We empirically investigated these failures and discussed the reasons in Section 4.2.2.

In summary, the contribution of this paper is threefold:

- We propose TTG-GEN, a method for generating TTG problems for LLM benchmarking using CGF. TTG-GEN can generate non-trivial problems at scale, offering a continuously updatable evaluation framework with a low risk of solution leakage for assessing repository-level code comprehension.
- We construct TTG-BENCH-LITE, a practical benchmark for evaluating repository-level input synthesis abilities in LLMs. It consists of 500 TTG problems created by applying TTG-GEN to 16 real-world C/C++ repositories that process well-known file formats, facilitating efficient evaluation.

108 • We conduct a comprehensive evaluation on TTG-BENCH-LITE with widely used LLMs
 109 under various settings. The results show that even the best-performing LLM can only
 110 achieve a pass@1 of 14.56%, indicating these tasks are a significant hurdle for current
 111 models and highlighting considerable room for improvement in LLMs.
 112

113 The code for TTG-GEN and TTG-BENCH-LITE, along with the containerized running environment,
 114 is currently available in supplementary materials and will be open-sourced after review.
 115

116 2 RELATED WORK 117

118 **LLM Benchmarks for Code Reasoning and Testing.** Recently, various benchmarks have been
 119 proposed to evaluate the capabilities of LLMs on code-related tasks and in practical software testing
 120 scenarios (Hudson et al., 2024; Koohestani et al., 2025). Several focus on function-level code com-
 121 prehension: CRUX-Eval (Gu et al., 2024) and its multilingual extension CRUX-Eval-X (Xu et al.,
 122 2024) assess input/output prediction for synthesized functions, while R-Eval (Chen et al., 2024)
 123 targets the inference of execution states. Another line of work evaluates unit test generation for real-
 124 world Python code. This includes SWT-bench (Mündler et al., 2024) for creating issue-reproducing
 125 tests, TestGen-Eval (Jain et al., 2025) for improving coverage, and CLOVER (Xu et al., 2025a) for
 126 meeting specific test objectives. Other benchmarks include Test-Eval (Wang et al., 2024) for gener-
 127 ating inputs to cover LeetCode problems and Test-Bench (Zhang et al., 2024a) for generating unit
 128 tests in Java.

129 **Coverage-Guided Fuzzing.** Coverage-guided fuzzing (CGF) is a prominent and highly effective
 130 automated technique for generating test inputs, widely applied to test complex, real-world systems
 131 (Zhu et al., 2022; Manès et al., 2019; Li et al., 2018). At its core, CGF employs an evolutionary
 132 algorithm that operates on a corpus of seed inputs, treating them as a population to be evolved (as
 133 shown in Figure 3 (b)). The key to CGF’s effectiveness is its feedback mechanism. The fuzzer
 134 monitors program execution to determine if the mutated input triggers new code coverage, such as
 135 executing previously unreached basic blocks, edges, or paths. If an input is deemed “interesting” for
 136 having increased coverage, it is retained and added to the seed pool for subsequent mutation rounds;
 137 otherwise, it is discarded. This simple yet powerful feedback loop continuously guides the search
 138 toward deeper and more complex program states. This technique has proven highly successful
 139 in practice, with widely adopted tools such as AFL (Google, 2015a), LibFuzzer (LLVM, 2015),
 140 and Honggfuzz (Google, 2014) making significant contributions to software security by uncovering
 thousands of vulnerabilities.

141 3 METHOD 142

143 3.1 THE TTG PROBLEM

146 3.1.1 TASK FORMULATION

147 **Problem Definition.** Formally, let R be a code repository, E be an entry program that takes a
 148 byte sequence (buffer) $b \in B$ as input, and L_{target} be a specific target code location within R .
 149 An instance of the targeted test-input generation (TTG) problem is a tuple $P = \langle R, E, L_{target} \rangle$.
 150 Let $exec(R, E, b)$ be a function that returns the set of code locations executed when program E is
 151 run with input buffer b . The objective is to find a byte sequence b such that the target location is
 152 executed: $L_{target} \in exec(R, E, b)$.
 153

154 **Solution Representation.** While the goal is to find the byte sequence b , LLMs often exhibit limi-
 155 tations in precise numerical calculations and direct binary data manipulation (Steyvers et al., 2025;
 156 Akhtar et al., 2023). Consequently, we do not require the model to output the raw byte sequence
 157 directly. Instead, the task is to generate a Python script S_{gen} which, when executed, produces the
 target byte sequence b .
 158

159 3.1.2 RATIONALE FOR TTG AS AN EVALUATION TASK 160

161 The TTG problem is an important task in software testing, reflecting a common scenario faced
 by developers during testing and debugging (Pezzè & Young, 2008; Clarke et al., 2018). From

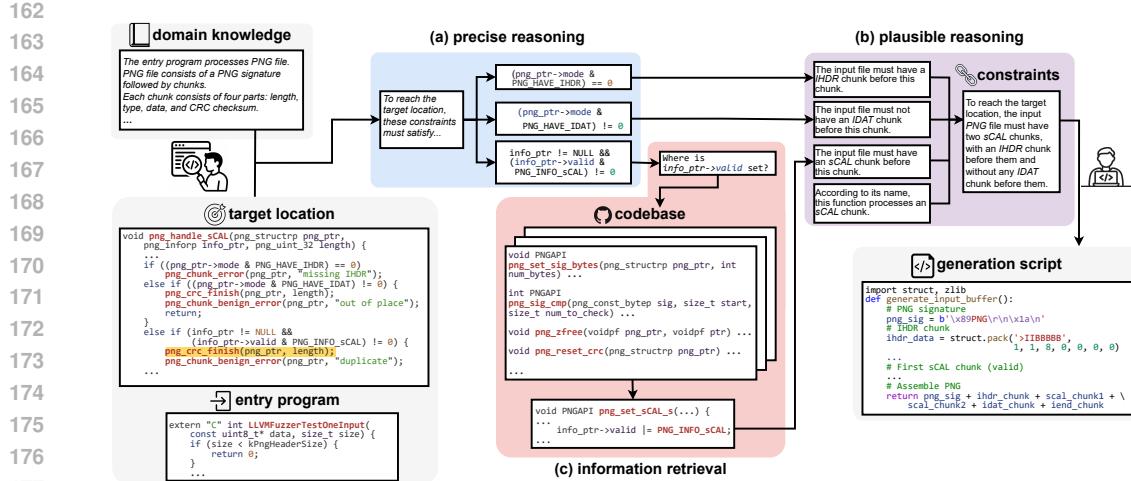


Figure 2: The reasoning process of a human analyst for a TTG problem. This process involves: (a) precise reasoning on code semantics, (b) plausible reasoning with domain knowledge and heuristics, and (c) information retrieval from the codebase.

a practical perspective, it is a core element of software testing, verification, and analysis. While this task is addressable by human experts (Brooks, 1983; Woodfield et al., 1981; Padioleau et al., 2009), traditional automated methods often face significant hurdles like path explosion (Baldoni et al., 2018; Pelánek, 2008). The capabilities of LLMs present a new avenue for tackling these long-standing issues by leveraging domain knowledge and heuristics. From a benchmarking perspective, TTG offers a robust framework for assessing a range of LLM capabilities in repository-level code comprehension, with success being unambiguously verifiable through execution. Furthermore, this approach enables automated generation of new problem instances at scale and carries a low risk of solution leakage, as the required inputs are not typically available in public data.

3.1.3 TASK MOTIVATION AND REQUIRED CAPABILITIES

To illustrate the capabilities required to solve such problems, we consider an example from *libpng*, where the goal is to execute a specific line within the *png_handle_sCAL* function (Figure 2).

An expert solves this by leveraging domain-specific knowledge, such as the *PNG* file format specifications, and performing a detailed analysis of the source code to trace control flow. The crucial step is mapping low-level code conditions to high-level properties of the input format, for instance, deducing that the presence and order of specific *PNG* chunks are required. This process of abstraction and constraint satisfaction enables the efficient construction of a valid input. As a benchmark, TTG is designed to assess a similar set of key abilities in LLMs, including: (1) Code Semantic Understanding, to correctly interpret program behavior; (2) Constraint Inference, to determine the precise conditions and execution paths needed to reach a target; (3) Abstract Mapping, to translate low-level code conditions into high-level input format constraints; and (4) Knowledge Application, to use domain-specific information to synthesize a valid input.

3.2 THE TTG-GEN METHOD

Motivation. While TTG problems are suitable for evaluating LLMs, selecting appropriate targets is critical for a meaningful benchmark. Randomly sampling locations is ineffective for several reasons. First, many locations in large code repositories are simply unreachable by a given entry program (Babić et al., 2019; Metzman et al., 2021). Second, many reachable locations, particularly in input parsing code, can be covered by simple random inputs and thus do not effectively test a model’s capabilities (Hamlet, 1994; Arcuri et al., 2011). Third, some targets may be reached by recalling standard file formats rather than by analyzing specific code paths.

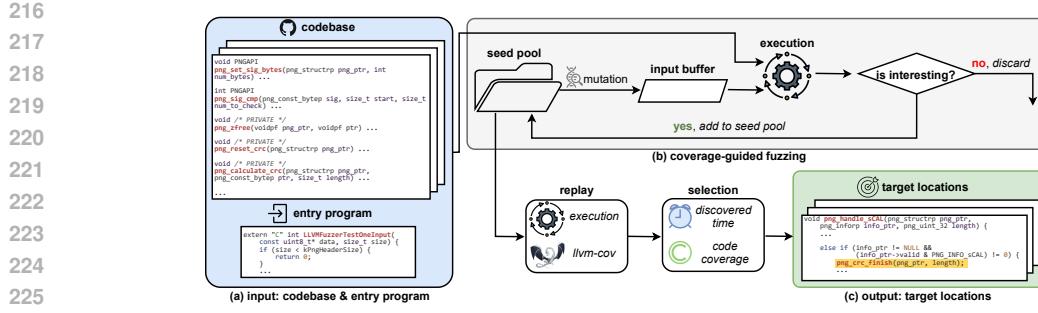


Figure 3: The workflow of TTG-GEN. The process takes (a) a codebase and its entry program as input, then applies (b) coverage-guided fuzzing to it; afterward, the generated seed inputs are replayed to select newly covered locations, ultimately producing (c) a set of curated target locations as output.

The Method. To construct a meaningful benchmark, the central task is to identify target locations that are demonstrably reachable yet non-trivial to cover. We find that coverage-guided fuzzing (CGF), a prominent automated software testing technique, is a suitable approach for this selection process. Our key insight is that code locations discovered in the later stages of a CGF campaign serve as ideal targets. The iterative nature of CGF provides three key properties: (1) Feasibility: a location’s discovery by the fuzzer confirms it is reachable; (2) Non-triviality: selecting later-discovered locations filters out targets easily covered by simple random inputs; (3) Structural Requirement: reaching these deeper locations often requires generating non-standard or subtly malformed inputs from an initial valid seed pool. This necessitates an analysis of complex code semantics and input structures rather than mere format recall, making the resulting problems well-suited for evaluation.

Algorithm 1 Generating Target Code Locations

```

Input: codebase, entryProgram, threshold
Output: a set of target locations
seeds, discoveredTime ← CoverageGuideFuzzing(codebase, entryProgram)
seeds ← sortByTime(seeds, discoveredTime)
coveredLocs ← ∅
candidateTargetLocs ← ∅
for seed ∈ seeds do
    seedCoveredLocs ← getSourceCoverage(codebase, entryProgram, seed)
    newlyCoveredLocs ← seedCoveredLocs \ coveredLocs
    if newlyCoveredLocs ≠ ∅ and discoveredTime[seed] ≥ threshold then
        targetLoc ← randomChoose(newlyCoveredLocs)
        candidateTargetLocs ← candidateTargetLocs ∪ {targetLoc}
    end if
    coveredLocs ← coveredLocs ∪ newlyCoveredLocs
end for
return candidateTargetLocs

```

The workflow of the proposed TTG-GEN is illustrated in Figure 3. The input to TTG-GEN is a real-world C/C++ repository with an accompanying entry program; the output is a curated set of target locations for constructing TTG problems.

The process begins by applying coverage-guided fuzzing (CGF) to the repository. TTG-GEN employs a hybrid fuzzing approach (Poeplau & Francillon, 2020), which combines a standard fuzzer with a symbolic execution engine to navigate complex path constraints. The initial seed pool is populated with valid input files (e.g., *PNG*, *JPEG*) from established corpora (Metzman et al., 2021), and TTG-GEN utilizes fuzzing dictionaries to guide the fuzzer towards syntactically valid inputs, thereby improving efficiency (Google, 2015b). After a set period of fuzzing, this process yields a corpus of generated inputs, each timestamped with its discovery time.

In the second stage, TTG-GEN analyzes the coverage generated by these inputs. The target program is first compiled with instrumentation flags to generate coverage data. TTG-GEN then replays the generated seed inputs chronologically and use the *llvm-cov* to process the resulting execution profiles (LLVM, 2025). A code location is selected as a candidate target if it is newly covered by a seed whose discovery time exceeds a predefined heuristic threshold (one hour in our experiments). This time-based filtering is designed to select for non-trivial targets that were not discoverable through simple mutations. The overall workflow is summarized in Algorithm 1.

Table 1: Comparison of TTG-GEN and other code reasoning / software testing benchmarks.

Benchmark	Task	Real-World	Code Scale	Programming Language	Scalable	Low Risk of Solution Leakage	Verifiable
CRUX-Eval	Input/Output Pred	✗	Function Level	Python	✓	✓	✓
CRUX-Eval-X	Input/Output Pred	✗	Function Level	Multiple PL	✓	✓	✓
R-Eval	Execution State Pred	✗	Function Level	Python	✗	✓	✓
SWT-Bench	Unit Test	✓	Repository Level	Python	✗	✗	✗
TestGen-Eval	Unit Test	✓	File Level	Python	✗	✗	✗
CLOVER	Unit Test	✓	Repository Level	Python	✗	✗	✗
Test-Eval	Input Generation	✗	File Level	Python	✗	✓	✓
Test-Bench	Unit Test	✓	Class Level	Java	✗	✗	✗
TTG-GEN (Ours)	Input Generation / System Test	✓	Repository Level	C/C++	✓	✓	✓

Limitations of Existing Benchmarks. While many benchmarks exist for evaluating LLMs on code-related tasks, they often exhibit shortcomings that limit their utility. Some are constructed from synthesized code snippets or isolated functions, which do not reflect the complexity of real-world software development. Others rely on static, human-authored problem specifications like GitHub issues or existing unit tests for problem creation and verification. This approach limits scalability, can lead to benchmark saturation, and introduces a risk of models recalling solutions seen during pre-training. Furthermore, the scope is often confined to small-scale Python code, leaving system-level testing of large C/C++ applications, which requires analyzing cross-module interactions, largely unexplored. Finally, some benchmarks employ verification methods that can inaccurately assess a model’s true capabilities by allowing incorrect solutions to pass existing test suites.

Advantages of Our Approach. The proposed TTG-GEN framework addresses these limitations by programmatically generating targeted test-input generation (TTG) problems from real-world C/C++ codebases. The TTG task is grounded in practical software testing and offers a precise, execution-based oracle for verification: a solution is correct if and only if the target location is executed. By leveraging coverage-guided fuzzing (CGF) for target selection, TTG-GEN enables the scalable generation of new problem instances with a low risk of solution leakage. This method ensures that the resulting benchmark is continuously updatable, grounded in realistic system-level contexts, and provides a robust measure of an LLM’s ability to comprehend and manipulate complex program behavior (see Table 1).

Table 2: Details of C/C++ repositories used in TTG-BENCH-LITE.

Repository	Description	Input File Type	#LOC
Bloaty	Program binary size analysis tool	Binary executable files	984k
FreeType2	Font rendering and processing library	Font files	441k
HarfBuzz	Font shaping engine	Font files	81k
LittleCMS	Color management system	Color profile files	69k
libjpeg-turbo	High-performance JPEG codec library	JPEG images	87k
libpng	PNG image decoding library	PNG images	138k
libxml2	XML parsing library	XML files	307k
OpenH264	H.264 video codec library	H264 video files	131k
OpenThread	Implementation of the Thread protocol	IPv6 network packets	521k
RE2	Efficient regular expression matching library	Regular expressions	38k
libsndfile	Audio file reading/writing library	Audio files	96k
SQLite3	Lightweight embedded database	SQLite database files	948k
stb	Simple image loading library	Image files	93k
libvorbis	Ogg Vorbis audio codec	Ogg Vorbis audio files	71k
woff2	WOFF2 font file converter	WOFF2 font files	57k
OpenSSL	Cryptography library with certificate handling	X.509 certificate files	645k

The TTG-BENCH-LITE Benchmark. Using TTG-GEN, we construct TTG-BENCH-LITE, a benchmark consisting of 500 TTG problems from real-world C/C++ programs to facilitate efficient evaluation. We select 16 widely-used real-world programs that process a wide range of well-known

324 file formats. The details of the programs are summarized in Table 2. To ensure a fair evaluation, we
 325 perform a preliminary check on all participating LLMs. We manually prompt each model to write
 326 Python scripts for generating valid files of the specified types and verify the outputs using format-
 327 specific parsers. This initial step confirms that the LLMs are capable of writing scripts to produce
 328 correctly formatted binary files and possess the necessary domain knowledge of the input formats.
 329 We run the fuzzer described for 24 hours, with the discovery time threshold set to one hour. From
 330 the resulting target locations returned by TTG-GEN, we randomly select 500 problems to form the
 331 problem set.

332

333 4 EVALUATION

334

335 4.1 SETTINGS

336

337 4.1.1 HANDLING LARGE-SCALE REPOSITORIES

338

339 Given a target code location, many parts of the code repository are semantically related to it. How-
 340 ever, the large scale of real-world code repositories makes it impractical to fit the entire repository
 341 into LLM’s context window. To address this, we adopt two standard settings, retrieval-based (Gao
 342 et al., 2023; Fan et al., 2024) and agent-based (Jin et al., 2024; Liu et al., 2024b; Yang et al., 2024),
 343 as described below. See Appendix A.2 for detailed settings.

344

345 Retrieval-Based Setting. In this paper, we opt to use BM25 (Robertson et al., 2009) as the retrieval
 346 method, following related works in repository-level benchmarking (Jimenez et al., 2023; Mündler
 347 et al., 2024). Given the target code location, we concatenate its surrounding code (within 10 lines)
 348 along with the signatures of related functions, classes, etc., to form the query string. The source
 349 code is split into syntax units (e.g., functions, classes, structs, methods) using Clang (Lattner &
 350 Adve, 2004). For syntax units longer than 100 lines, we further split them until every units are less
 351 than or equal to 100 lines. Code comments are retained as they are crucial for understanding the
 352 semantic of code snippets, as discussed in Section 3.1.2. The query strings and splitted syntax units
 353 are tokenized using Clang. We then apply a standard BM25 retrieval approach to retrieve relevant
 354 syntax units, ensuring they fits within the context length of 15000 tokens.

355

356 Agent-Based Setting. Agent-based methods (Jin et al., 2024; Liu et al., 2024b) are commonly
 357 used for repository-level software engineering tasks, where LLM agents are equipped with tools to
 358 browse, search, and edit code. However, the tools provided to agents in prior works are typically
 359 focused on issue resolution or unit test generation tasks (Yang et al., 2024; Zhang et al., 2024b) and
 360 may not be as effective for solving the TTG problem. To address this, we design a code browsing
 361 tool with functionalities tailored specifically to the needs of the TTG problem, drawing from routines
 362 frequently used by human analysts, such as query all code snippets where a given function is called
 363 (details see Table 5 in Appendix). The usage of the tool is demonstrated in prompt with concrete
 364 examples. Moreover, the agent is allowed to emit multiple queries to the tool in a single session of
 365 conversation, helping to reduce the number of conversation rounds to improve efficiency.

366

367 4.1.2 LARGE LANGUAGE MODELS

368

We consider O3-mini (OpenAI, 2025b), GPT-4o-mini (OpenAI, 2025a), DeepSeek-R1 (Guo et al.,
 369 2025), DeepSeek-V3 (Liu et al., 2024a), Qwq (Qwen, 2025b), Qwen series (Qwen, 2025a; 2024),
 370 Gemma-3 (Team et al., 2025), Mistral-3.1 (Mistral, 2025), Cogito-V1 (Cogito, 2025), Llama-3.3
 371 (Grattafiori et al., 2024), and GLM-4 (GLM et al., 2024) as the underlying LLMs in both retrieval-
 372 based and agent-based settings. Among these, GPT-4o-mini, DeepSeek-V3, Gemma-3, Mistral-
 373 3.1, Qwen-2.5, Llama-3.3, and GLM-4 are standard LLMs, while O3-mini, DeepSeek-R1, Qwq,
 374 R1-Distill-Qwen and GLM-Z1 are reasoning LLMs (Plaat et al., 2024; Li et al., 2025). Qwen-3
 375 and Cogito are hybrid reasoning models. For these models, we set the parameters to the values
 376 recommended by the respective model providers. See Appendix A.1 for more details.

377

378 4.2 RESULTS

379

380 For each LLM, the pass@1-5 scores under both retrieval-based and agent-based settings are reported
 381 in Table 3. Overall, the performance of different LLMs varies widely, demonstrating that TTG-

378

379

Table 3: The pass@1-5 scores (%) of LLMs on TTG-BENCH-LITE.

Model	Size	Thinking Mode	Reasoning	Retrieval-based Pass@k					Agent-based Pass@k				
				k = 1	k = 2	k = 3	k = 4	k = 5	k = 1	k = 2	k = 3	k = 4	k = 5
O3-mini	N/A		✓	12.00	15.20	16.62	17.44	18.00	11.96	15.90	18.06	19.56	20.80
GPT-4o-mini	N/A		✗	0.56	0.84	1.06	1.24	1.40	0.56	0.72	0.88	1.04	1.20
DeepSeek-R1	671B		✓	14.56	18.36	20.48	21.96	23.00	14.44	19.38	21.84	23.40	24.60
DeepSeek-V3	671B		✗	7.68	11.08	13.04	14.48	15.60	8.28	11.88	14.12	15.64	16.80
Llama-3.3	70B		✗	2.20	3.24	3.88	4.28	4.60	1.52	2.54	3.32	3.92	4.40
Qwq	32B		✓	7.20	10.78	12.84	14.20	15.20	5.76	9.78	12.62	14.68	16.20
Qwen-3	32B	On	✓	6.16	8.84	10.52	11.64	12.40	5.88	8.32	9.84	10.96	11.80
Qwen-3	32B	Off	✗	1.24	1.96	2.54	3.08	3.60	2.96	4.74	6.00	7.00	7.80
Qwen-3	30B-A3B	On	✓	3.24	4.60	5.34	5.84	6.20	2.36	3.62	4.54	5.20	5.60
Qwen-3	30B-A3B	Off	✗	0.68	1.22	1.64	1.96	2.20	0.76	1.20	1.48	1.68	1.80
Qwen-2.5	32B		✗	1.56	2.38	2.90	3.28	3.60	2.16	3.18	3.90	4.48	5.00
Gemma-3	27B		✗	0.32	0.62	0.90	1.16	1.40	0.88	1.50	1.96	2.32	2.60
Mistral-3.1	24B		✗	0.04	0.08	0.12	0.16	0.20	0.16	0.30	0.42	0.52	0.60
Cogito-V1	32B	On	✓	1.52	2.72	3.68	4.48	5.20	2.48	4.26	5.50	6.32	6.80
Cogito-V1	32B	Off	✗	1.12	1.94	2.66	3.28	3.80	1.24	2.10	2.76	3.32	3.80
R1-Distill-Qwen	32B		✓	2.84	4.70	6.02	6.96	7.60	1.80	2.98	3.80	4.44	5.00
GLM-4	32B		✗	0.96	1.80	2.54	3.20	3.80	1.52	2.66	3.54	4.24	4.80
GLM-Z1	32B		✓	2.24	3.36	4.14	4.80	5.40	2.20	3.36	4.18	4.84	5.40

392

393

BENCH-LITE has good discriminability, which is crucial for benchmarking. The best-performing model (DeepSeek-R1) achieves a pass@1 score of 14.56% and a pass@5 score of 24.60%, and most median-sized LLMs can achieve only pass@1 scores of 0.04%-7.20% and pass@5 scores of 0.20%-16.20%, indicating that TTG-BENCH-LITE is challenging for state-of-the-art LLMs.

398

399

4.2.1 OBSERVATIONS

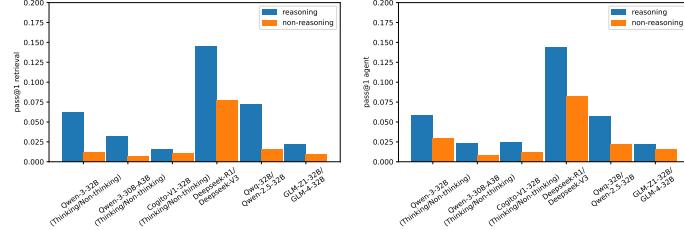


Figure 4: Comparison between reasoning LLMs with their non-reasoning counterparts.

410

411

Comparison of reasoning and non-reasoning models. To assess the impact of reasoning capabilities gained from reinforcement learning (Plaat et al., 2024; Li et al., 2025) on LLMs’ performance in solving the TTG problem, we compare the pass@1 scores of different reasoning LLMs with their non-reasoning counterparts. Six pairs of LLMs are used for this comparison: hybrid reasoning models with the reasoning option on and off (Qwen-3-32B, Qwen-3-30B-A3B, and Cogito-V1), and reasoning versus non-reasoning models that share the same base architecture (DeepSeek-R1/DeepSeek-V3, Qwq-32B/Qwen-2.5-32B, GLM-Z1-32B/GLM-4-32B). Figure 4 compares the pass@1 scores for these models. Across all six pairs, reasoning models outperform their non-reasoning counterparts by a large margin, indicating that the reasoning ability acquired through RL significantly enhances performance in solving TTG problems.

421

422

Comparison of retrieval-based and agent-based settings. The performance under retrieval-based and agent-based settings varies across different models. The pass@1 scores of various models under both settings are visualized in Figure 5, with reasoning models shown on the left of the dashed line and non-reasoning models on the right. In general, for pass@1, the retrieval-based setting tends to yield better performance for most reasoning models, while the agent-based setting provides superior results for most non-reasoning models. From our manual examinations, we speculate that this is because (1) reasoning models excel at handling long contexts by generating longer chains of thought, which allows them to better reason about complex relationships within the code, and (2) reasoning models are more prone to hallucination, which makes them less effective at using the tools provided in the agent-based setting. Detailed analyses can be found in Appendix D.1.

431

Performance of LLMs across different repositories. The pass@1 scores of top-performing LLMs across 16 C/C++ repositories are shown in Figure 6 in Appendix. The scores vary dramatically, with

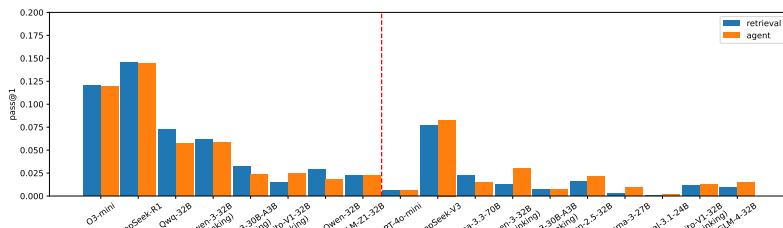


Figure 5: Pass@1 scores of LLMs under different settings.

some repositories achieving near 50%, while others are close to 0%. This variation can likely be attributed to the inherent differences in the complexity of the repositories and their entry programs, and suggests that current LLMs may struggle with reasoning in more complex contexts, where intricate code structures and dependencies are involved.

Solved problems by different LLMs. We analyzed the problems solved by different LLMs across all 500 problems in 5 trials, under both retrieval-based and agent-based settings. The result can be found in Appendix D.2, which highlights that, despite differing overall performance, many LLMs are able to solve a unique set of problems that the others cannot. This demonstrates the complementary strengths of each model in addressing various aspects of the TTG problem.

4.2.2 CASE STUDIES

We manually examined some of the LLMs’ responses to solve the TTG problems. In some cases, LLMs performed well, demonstrating impressive abilities in precise reasoning about path conditions and plausible reasoning about the connection between path conditions and high-level file formats, which are similar to human analysts. However, in many cases, LLMs failed to reach the target location, and we identified several reasons for these failures: First, many LLMs (especially reasoning models) tend to hallucinate and rely on fabricated code snippets for decision-making rather than querying (in the agent-based setting). While these fabricated code snippets may seem plausible, they are not contextually appropriate and not reflective of the real codebase. Second, although LLMs have detailed knowledge about valid target file format specifications (examined by manual prompting), they often struggle to synthesize the correct file format (e.g., a valid *IDAT* chunk in a *PNG* file). Sometimes, they miss key specifications, highlighting the difference between knowledge and their application for LLMs. Third, in some instances, LLMs fail to make plausible connections between low-level conditions and high-level constraints on file formats. As a result, they are unable to generate correct test inputs. Fourth, for complex code reasoning scenarios, where constraints are distributed across a broader context rather than within a single function, LLMs sometimes overlook important constraints. These observations suggest that LLMs still have a large room for improvement and are far from achieving the level of expertise exhibited by human analysts. Detailed analyses and examples can be found in Appendix C.

5 CONCLUSION

Evaluating Large Language Models (LLMs) on repository-level code comprehension tasks is crucial for understanding their capabilities and applying them to practical software testing. However, existing benchmarks often rely on static, human-authored data for problem creation and verification, which limits scalability and introduces a risk of solution leakage. In this paper, we present TTG-GEN, an automated method for generating targeted test-input generation (TTG) problems representative of scenarios faced by developers. By utilizing Coverage-Guided Fuzzing (CGF), TTG-GEN generates non-trivial and reachable TTG problems from real-world C/C++ repositories, ensuring scalability with a low risk of solution leakage. We applied TTG-GEN to 16 real-world C/C++ packages to construct TTG-BENCH-LITE, a dataset of 500 problems. Our evaluation shows that even the best-performing LLMs can solve only a small fraction of these problems, indicating that system-level code comprehension remains a significant hurdle for current models.

486

6 REPRODUCIBILITY STATEMENT

488 In the supplementary material, we provide the code of TTG-GEN and the TTG-BENCH-LITE
 489 dataset, with a *README* file for illustrating their usage and how to reproduce the results. We
 490 have uploaded the supplementary material to the submission site (OpenReview). We believe it can
 491 guarantee the reproducibility of our experiments.

493

REFERENCES

495 Mubashara Akhtar, Abhilash Shankarampeta, Vivek Gupta, Arpit Patil, Oana Cocarascu, and Elena
 496 Simperl. Exploring the numerical reasoning capabilities of language models: A comprehensive
 497 analysis on tabular data. In *The 2023 Conference on Empirical Methods in Natural Language
 498 Processing*, 2023. URL <https://openreview.net/forum?id=1qJgZUAc8j>.

499 Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results
 500 and practical implications. *IEEE transactions on Software Engineering*, 38(2):258–277, 2011.

501 Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline
 502 Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *Proceed-
 503 ings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and
 504 Symposium on the Foundations of Software Engineering*, pp. 975–985, 2019.

505 Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A
 506 survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

507 Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed grey-
 508 box fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communica-
 509 tions security*, pp. 2329–2344, 2017.

510 Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal
 511 of man-machine studies*, 18(6):543–554, 1983.

512 Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning Runtime Behavior of a Program with LLM: How Far Are We?
 513 March 2024. URL <https://www.semanticscholar.org/paper/f11cbd0d24b0e3fb917fe14cfaf572e76402e5df>.

514 Yu Cheng, Yi Chang, and Yuan Wu. A survey on data contamination for large language models.
 515 *ArXiv*, abs/2502.14425, 2025.

516 Edmund M Clarke, Thomas A Henzinger, and Helmut Veith. Introduction to model checking. *Hand-
 517 book of Model Checking*, pp. 1–26, 2018.

518 Team Cogito. Introducing cogito preview. <https://www.deepcogito.com/research/cogito-v1-preview>,
 519 2025. Accessed: 2025-06-01.

520 Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. Generalization or
 521 memorization: Data contamination and trustworthy evaluation for large language models, 2024.
 522 URL <https://arxiv.org/abs/2402.15938>.

523 Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and
 524 Qing Li. A survey on rag meeting llms: Towards retrieval-augmented large language models. In
 525 *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*,
 526 pp. 6491–6501, 2024.

527 Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun,
 528 Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A
 529 survey. *arXiv preprint arXiv:2312.10997*, 2:1, 2023.

530 Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu
 531 Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng,
 532 Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu,
 533

534 Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu
 535 Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng,
 536 Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu,
 537

538 Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu
 539 Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng,
 Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu,

540 Minlie Huang, Peng Zhang, Qinkai Zheng, Rui Lu, Shuaiqi Duan, Shudan Zhang, Shulin Cao,
 541 Shuxun Yang, Weng Lam Tam, Wenyi Zhao, Xiao Liu, Xiao Xia, Xiaohan Zhang, Xiaotao Gu,
 542 Xin Lv, Xinghan Liu, Xinyi Liu, Xinyue Yang, Xixuan Song, Xunkai Zhang, Yifan An, Yifan
 543 Xu, Yilin Niu, Yuantao Yang, Yueyan Li, Yushi Bai, Yuxiao Dong, Zehan Qi, Zhaoyu Wang,
 544 Zhen Yang, Zhengxiao Du, Zhenyu Hou, and Zihan Wang. Chatglm: A family of large language
 545 models from glm-130b to glm-4 all tools, 2024.

546 Google. Honggfuzz — honggfuzz. <https://honggfuzz.dev/>, 2014. Accessed: 2025-06-01.

547 Google. American fuzzy lop. <https://lcamtuf.coredump.cx/afl>, 2015a. Accessed: 2025-06-01.

548 Google. Afl fuzzing dictionaries. <https://github.com/google/AFL/tree/master/dictionaries>, 2015b.
 549 Accessed: 2025-06-01.

550 Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad
 551 Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd
 552 of models. *arXiv preprint arXiv:2407.21783*, 2024.

553 Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I
 554 Wang. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. 2024.

555 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu,
 556 Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms
 557 via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

558 Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 2:971–978, 1994.

559 Xinying Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo,
 560 John C. Grundy, and Haoyu Wang. Large language models for software engineering: A systematic
 561 literature review. *ACM Trans. Softw. Eng. Methodol.*, 33:220:1–220:79, 2023.

562 Sinclair Hudson, Sophia Jit, Boyue Caroline Hu, and Marsha Chechik. A software engi-
 563 neering perspective on testing large language models: Research, practice, tools and bench-
 564 marks. *ArXiv*, abs/2406.08216, 2024. URL <https://api.semanticscholar.org/CorpusID:270391588>.

565 Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. TestGenEval: A Real World Unit Test Genera-
 566 tion and Test Completion Benchmark, March 2025. URL <http://arxiv.org/abs/2410.00752>. arXiv:2410.00752 [cs].

567 Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language
 568 models for code generation. *ArXiv*, abs/2406.00515, 2024.

569 Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik
 570 Narasimhan. Swe-bench: Can language models resolve real-world github issues? *ArXiv*,
 571 abs/2310.06770, 2023.

572 Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to
 573 llm-based agents for software engineering: A survey of current, challenges and future. *ArXiv*,
 574 abs/2408.02479, 2024.

575 Roham Kohestani, Philippe de Bekker, and Maliheh Izadi. Benchmarking AI Models in Software
 576 Engineering: A Review, Search Tool, and Enhancement Protocol. 2025. doi: 10.48550/ARXIV.
 577 2503.05860. URL <https://arxiv.org/abs/2503.05860>. Publisher: arXiv Version
 578 Number: 1.

579 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis &
 580 transformation. In *International symposium on code generation and optimization*, 2004. *CGO*
 581 2004., pp. 75–86. IEEE, 2004.

582 Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.

594 Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian
 595 Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, Yingying Zhang, Fei Yin, Jiahua Dong, Zhiwei
 596 Li, Bao-Long Bi, Ling-Rui Mei, Junfeng Fang, Zhijiang Guo, Le Song, and Cheng-Lin Liu.
 597 From system 1 to system 2: A survey of reasoning large language models, 2025. URL <https://arxiv.org/abs/2502.17419>.

598

599 Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao,
 600 Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint*
 601 *arXiv:2412.19437*, 2024a.

602

603 Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling
 604 Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint*
 605 *arXiv:2409.02977*, 2024b.

606 Team LLVM. Libfuzzer: A library for coverage-guided fuzz testing.
 607 <https://llvm.org/docs/LibFuzzer.html>, 2015. Accessed: 2025-06-01.

608

609 Team LLVM. llvm-cov - emit coverage information. <https://llvm.org/docs/CommandGuide/llvm-cov.html>, 2025. Accessed: 2025-06-01.

610

611 Sanoop Malliserry and Yu-Sung Wu. Demystify the fuzzing methods: A comprehensive survey.
 612 *ACM Computing Surveys*, 56(3):1–38, 2023.

613

614 Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J
 615 Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE*
 616 *Transactions on Software Engineering*, 47(11):2312–2331, 2019.

617 Jonathan Metzman, László Szekeres, Laurent Simon, Read Spraberry, and Abhishek Arya.
 618 Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM*
 619 *joint meeting on European software engineering conference and symposium on the foundations*
 620 *of software engineering*, pp. 1393–1403, 2021.

621

622 Team Mistral. Mistral small 3.1. <https://mistral.ai/news/mistral-small-3-1>, 2025. Accessed: 2025-
 623 06-01.

624 Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin T. Vechev. Swt-bench: Testing and
 625 validating real-world bug-fixes with code agents. In *Neural Information Processing Systems*,
 626 2024.

627

628 OpenAI. Gpt-4o mini: advancing cost-efficient intelligence. <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence>, 2025a. Accessed: 2025-06-01.

629

630 OpenAI. Openai o3-mini. <https://openai.com/index/openai-o3-mini>, 2025b. Accessed: 2025-06-01.

631

632 Yoann Padolieau, Lin Tan, and Yuanyuan Zhou. Listening to programmers—taxonomies and char-
 633 acteristics of comments in operating system code. In *2009 IEEE 31st International Conference*
 634 *on Software Engineering*, pp. 331–341. IEEE, 2009.

635 Radek Pelánek. Fighting state space explosion: Review and evaluation. In *International Workshop*
 636 *on Formal Methods for Industrial Critical Systems*, pp. 37–52. Springer, 2008.

637

638 Mauro Pezzè and Michal Young. *Software testing and analysis: process, principles, and techniques*.
 639 John Wiley & Sons, 2008.

640 Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki van Stein, and Thomas Back. Rea-
 641 soning with large language models, a survey. *arXiv preprint arXiv:2407.11511*, 2024.

642

643 Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret,
 644 compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pp. 181–198, 2020.

645

646 Team Qwen. Qwen2.5: A party of foundation models, September 2024. URL <https://qwenlm.github.io/blog/qwen2.5/>.

647

648 Team Qwen. Qwen3, April 2025a. URL <https://qwenlm.github.io/blog/qwen3/>.

648 Team Qwen. Qwq-32b: Embracing the power of reinforcement learning, March 2025b. URL
 649 <https://qwenlm.github.io/blog/qwq-32b/>.
 650

651 Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and be-
 652 yond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

653 Mark Steyvers, Heliodoro Tejeda, Aakriti Kumar, Catarina Belem, Sheer Karny, Xinyue Hu,
 654 Lukas W Mayer, and Padhraic Smyth. What large language models know and what people think
 655 they know. *Nature Machine Intelligence*, pp. 1–11, 2025.

656

657 Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej,
 658 Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical
 659 report. *arXiv preprint arXiv:2503.19786*, 2025.

660 Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. Sok: The progress, challenges, and
 661 perspectives of directed greybox fuzzing. *arXiv preprint arXiv:2005.11907*, 2020.

662

663 Wenhao Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming
 664 Zhang, An Ran Chen, and Lei Ma. TESTEVAL: Benchmarking Large Language Models for Test
 665 Case Generation. 2024.

666 Scott N Woodfield, Hubert E Dunsmore, and Vincent Y Shen. The effect of modularization and
 667 comments on program comprehension. In *Proceedings of the 5th international conference on*
 668 *Software engineering*, pp. 215–223, 1981.

669

670 Jiacheng Xu, Bo Pang, Jin Qu, Hiroaki Hayashi, Caiming Xiong, and Yingbo Zhou. CLOVER:
 671 A Test Case Generation Benchmark with Coverage, Long-Context, and Verification, February
 672 2025a. URL <http://arxiv.org/abs/2502.08806>. arXiv:2502.08806 [cs].

673 Junjielong Xu, Qinan Zhang, Zhiqing Zhong, Shilin He, Chaoyun Zhang, Qingwei Lin, Dan Pei,
 674 Pinjia He, Dongmei Zhang, and Qi Zhang. OPENRCA: CAN LARGE LANGUAGE MODELS
 675 LO- CATE THE ROOT CAUSE OF SOFTWARE FAILURES? 2025b.

676

677 Ruiyang Xu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and
 678 Le Sun. CRUXEval-X: A Benchmark for Multilingual Code Reasoning, Understanding and Ex-
 679 ecution, August 2024. URL <http://arxiv.org/abs/2408.13001>. arXiv:2408.13001
 680 [cs].

681 John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan,
 682 and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineer-
 683 ing, November 2024. URL <http://arxiv.org/abs/2405.15793>. arXiv:2405.15793
 684 [cs].

685 Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. TestBench:
 686 Evaluating Class-Level Test Case Generation Capability of Large Language Models, September
 687 2024a. URL <http://arxiv.org/abs/2409.17561>. arXiv:2409.17561 [cs].

688

689 Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous
 690 program improvement, 2024b. URL <https://arxiv.org/abs/2404.05427>.

691

692 Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui
 693 Wang. A survey on language models for code. *ArXiv*, abs/2311.07989, 2023.

694

695 Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM*
 696 *Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

697

698

699

700

701

Model	Type	Description
O3-mini	Reasoning-focused	Specialized in complex reasoning tasks, optimized for logical inference.
GPT-4o-mini	General-purpose	A smaller variant of GPT-4o, balancing performance and efficiency.
DeepSeek-R1	Reasoning-focused	Designed for enhanced reasoning capabilities, particularly in structured tasks.
DeepSeek-V3	General-purpose	A versatile model with broad applicability across NLP tasks.
Qwq	Reasoning-focused	Optimized for step-by-step reasoning and problem-solving.
Qwen-3	Hybrid	Combines general language understanding with specialized reasoning abilities.
Qwen-2.5	General-purpose	A comprehensive series LLMs designed to meet diverse needs.
Gemma-3	General-purpose	A compact yet powerful open-weight model by Google DeepMind.
Mistral-3.1	General-purpose	Efficient and high-performing, suitable for diverse NLP applications.
Cogito-V1	Hybrid	Integrates general language modeling with structured reasoning mechanisms.
Llama-3.3	General-purpose	Meta's open-weight model, optimized for scalability and task generalization.
GLM-4	General-purpose	Supporting multilingual understanding and generation, suitable for a wide range of NLP tasks.
GLM-Z1	Reasoning-focused	Specializes in advanced decision-making and complex reasoning tasks.

A DETAILED EVALUATION SETTINGS

A.1 LARGE LANGUAGE MODELS

We list the LLMs used for evaluation in Table 4. For closed-source LLMs, we run them by calling APIs provided by the model developers. For open-source LLMs, we deploy them locally on an Ubuntu 22.04.2 LTS machine equipped with Intel Xeon Gold 6338 processors and NVIDIA H100 PCIe GPUs.

A.2 RETRIEVAL-BASED SETTING

A.2.1 RETRIVING WITH BM25

To facilitate retrieval using BM25, we divide the source code related to the target location into chunks. We only consider the C/C++ source files that participate in the compilation of the target fuzz driver. For programming languages like C/C++, it is natural to split source code into syntax units for retrieval to preserve semantic integrity. To achieve this, we use Clang to parse the source code into abstract syntax trees (ASTs), where subtrees correspond to syntax units. Initially, we split the code into high-level syntax units (e.g., function declarations, class declarations, method declarations, struct declarations, enum declarations, etc.). For syntax units containing more than 100 lines, we further split them until each unit contains fewer than or equal to 100 lines of code. We then extract the surrounding code of the target location, which is limited to 10 lines or fewer. Next, we extract related function signatures, class definitions, etc., and concatenate them with the surrounding code and the signature of the entry function LLVMFuzzerTestOneInput to form the query string. The tokenization process is carried out using standard C/C++ lexical analysis. We retrieve code chunks based on the query string, calculate their BM25 scores, and sort them according to the scores. Finally, we select as many of the top-scoring chunks as possible, as long as they fit within a 15,000-token limit.

A.2.2 PROMPT TEMPLATES

We list the prompt templates used in this section. We adopt a one-shot prompt strategy to facilitate effective interactions with the LLMs.

System Prompt. The system prompt consists of an objective description and an exemplary interaction. The objective description is shown in Listing A.2.2, and the exemplary interaction is shown in Listing A.2.2.

Listing 1: Objective description in retrieval-based setting

```
You are a software analysis expert. Your goal is to analyze source code to generate input
→ byte sequences for C/C++ programs, allowing the program to execute and reach specific
→ code locations when provided with these sequences.
```

Objective Description

```

756
757 Note that the program may undergo a series of processing and checks before reaching the
758 → specified line while handling the input byte sequence. You need to ensure that all these
759 → checks pass smoothly.
760 The input byte sequence for the program will be passed to the data parameter of the
761 → function:
762   ````C
763   int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)
764   ````

765 Here, size represents the length of the input bytes. You need to ensure that after reading
766 → the input from the LLVMFuzzerTestOneInput function, the program can execute and reach
767 → the specified code line in the target function. Therefore, you may need to analyze the
768 → call chain from LLVMFuzzerTestOneInput to the target function.
769 Note: Do not make any assumptions, do not use "possibly"; all information must be derived
770 → from known background knowledge or obtained through code analysis.
771
772 ##### Format for Returning Input Sequence
773 If you believe that you can find the input byte sequence without further queries, please
774 → write a Python function named generate_buf. When this function is called, it will
775 → randomly generate a byte sequence that meets the requirements.
776 You can call external Python packages within this function to perform specific numerical
777 → calculations. Please include the code for this function and any relevant auxiliary
778 → functions in the code field of the response JSON, and place the call chain from
779 → LLVMFuzzerTestOneInput to the target function in the call_chain field.
780 For example:
781   ````json
782   {
783     "code": "import random\ndef generate_buf():\n    frag_num = 1\n    frag_total = 1\n    session_id = random.randint(0, 99999)\n    payload = b'payload'\n    data_part =\n    f\"AAAAA,{frag_num},{frag_total},{session_id},{payload}\"\n    full_sentence =\n    f\"!{data_part}*\\u0007\"\n    return full_sentence.encode('ascii')",
784     "call_chain": ["LLVMFuzzerTestOneInput", "func1", "func2", "target_func"]
785   }
786   ````

787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809

```

Listing 2: The exemplary interaction in system prompt

```

785 ##### Session Example
786 An example of a session process is as follows:
787 User:
788 The program point I need to reach is the first line of the following code:
789   ````C
790   return 1;
791   ````

792 This code is located in the function func, and the expanded version of this function is as
793 → follows:
794   ````C
795   int func(uint8_t ch) {
796     if (ch == 'A'){
797       return 1;
798     }
799     return 0;
800   }
801   ````

802 The related code snippets is as follows:
803   ````C
804   int func(uint8_t ch) {
805     if (ch == 'A'){
806       return 1;
807     }
808     return 0;
809   }
810   int handle_file(const uint8_t* data, size_t size) {
811     if (size > 0){
812       return func(data[0]);
813     }
814     else {
815       return 0;
816     }
817   }
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910

```

```

810
811 }
812 int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
813     handle_file(data, size);
814 }
815
816 Assistant:
817 To reach this branch, the variable ch needs to equal 'A'.
818 By analyzing the code, the variable ch comes from the first element of the data array, and
819 → we must ensure that size > 0.
820 The current function call chain to reach the target code is:
821 LLVMFuzzerTestOneInput -> handle_file -> func.
822 In this chain:
823 LLVMFuzzerTestOneInput calls handle_file via
824
825     handle_file(data, size);
826
827
828 handle_file then calls func via
829
830     if (size > 0){
831         return func(data[0]);
832     }
833
834
835 In func, it reaches the target line through
836
837     if (ch == 'A'){
838         return 1;
839     }
840
841 and all the if conditions must pass.
842
843 From the analysis, the variable ch in the func function comes from the first element of the
844 → data in the handle_file function. The data and size in handle_file are directly passed
845 → from LLVMFuzzerTestOneInput.
846 Thus, I can conclude that the constraints to reach the target code are:
847 The length of the input byte sequence must be > 0.
848 The first element of the input byte sequence must be 'A'.
849 The input string b'A' meets these requirements.
850 Therefore, the final answer is:
851
852     json
853     {
854         "code": "def generate_buf():\n            return b'A'\n",
855         "call_chain": ["LLVMFuzzerTestOneInput", "handle_file", "func"]
856     }
857
858
859
860
861
862
863

```

User Prompt. The user prompt template is shown in Listing A.2.2, where "code" refers to the 5 lines surrounding the target location, "surrounding_code" refers to the 10 lines surrounding the target location, and "context" refers to the related code snippets retrieved using BM25.

Listing 3: User prompt template in retrieval-based setting

```

850
851
852 The point I need to reach is the first line of the following code:
853
854     code
855
856 It is only necessary to execute up to the first line of this code, without considering its
857 → content and execution context.
858 To facilitate locating this code, below is a snippet of the surrounding code:
859
860     surround_code
861
862
863 The context code is as follows:
864
865     context
866
867 This code is in C/C++ package {prog_package}.

```

864

865 Table 5: The specialized tool provided in the agent-based setting.

866 Input	867 Description
868 Function name	869 Query the code of the given function
870 Class name, Method name	871 Query the code of the given class method
872 Function name	873 Query all code snippets where the function is called
874 Class name, Method name	875 Query all code snippets where the method is called
876 Global variable name	877 Query the definition of the global variable
878 Global variable name	879 Query all code snippets that access the global variable
880 Class/struct member variable	881 Query all code snippets that access the member variable
882 Class/struct/enum type name	883 Query the definition of the class/struct/enum type
884 Macro name	885 Query the macro definition

874

875

876 A.3 AGENT-BASED SETTING

877

878 A.3.1 TOOL SPECIFICATION AND IMPLEMENTATION

879

880 The tools provided in related SE agent works are primarily designed for tasks such as issue resolution
 881 and unit test generation, which are not well-suited for TTG problems and often incur a significant
 882 cost in interactions between LLMs. To address this, we design a code browser with functionalities
 883 tailored specifically for TTG problems, drawing inspiration from routines commonly used by human
 884 analysts. The functionalities of the code browser are shown in Table 5. We demonstrate the usage
 885 of the code browser through prompts with concrete examples, utilizing a multi-round conversation.
 886 In this setup, LLMs query the code browser to browse relevant code snippets, which enhances their
 887 ability to solve TTG tasks.

888

889

890 A.3.2 PROMPT TEMPLATES

891

892

893 We list the prompt templates used in this section. We adopt a one-shot prompt strategy to facilitate
 894 effective interaction with LLMs.

895

896 **System Prompt.** The system prompt consists of the objective description, tool specification, and an
 897 exemplary interaction. The objective description is shown in Listing A.3.2. The tool specification is
 898 shown in Listing A.3.2. The exemplary interaction is shown in Listing A.3.2.

899

900 Listing 4: Objective description in agent-based setting

901

```
902 You are a software analysis expert. Your overall goal is to analyze source code to generate
  → input byte sequences for C/C++ programs, allowing the program to execute and reach
  → specific code locations when provided with these sequences.
```

903

904 In a single session, you need to make one of the following two choices:

- 905 1. If the information is insufficient, return the information you need to inquire about.
- 906 2. If the information is sufficient to make a decision, return the input sequence.

907

908 Please note that you should choose option 2 only when you are certain that the information
 → is adequate and that the input sequence can be completely determined. Otherwise, you
 → should choose option 1 to obtain more relevant information.

909

910 **#### Objective Description**
 911 Note that the program may undergo a series of processing and checks before reaching the
 → specified line while handling the input byte sequence. You need to ensure that all these
 → checks pass smoothly.

912

913 The input byte sequence for the program will be passed to the data parameter of the
 → function:

```
914 ````C
```

```
915 int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size)
  ````
```

916

917 Here, size represents the length of the input bytes. You need to ensure that after reading
 → the input from the LLVMFuzzerTestOneInput function, the program can execute and reach
 → the specified code line in the target function. Therefore, you may need to analyze the
 → call chain from LLVMFuzzerTestOneInput to the target function.

```

918
919 Be aware that you must perform program analysis using known information (such as call chain
920 → analysis, path analysis through if conditions, etc.) to guarantee that the provided
921 → input allows execution to reach the specified code line in the target function from the
922 → LLVMFuzzerTestOneInput function. If this cannot be assured, you should use the code
923 → browser to acquire more information until you can guarantee this.
924
925 You have the following choices:
926 1. When the context or function call information provided is insufficient to generate a
927 → suitable byte sequence, you can use the code browser to browse related code snippets to
928 → obtain more context information or function call information. The user will return the
929 → code snippets you request, and you can analyze the code based on this information to
930 → make further decisions. If these new code snippets are still insufficient to generate
931 → the required sequence, you can initiate another request. Please do not assume function
932 → or structure content or input processing conditions.
933 2. If you believe the information provided is sufficient, you can give the answer directly.
934 → In your final answer, you should explain the call chain from the LLVMFuzzerTestOneInput
935 → function to the target function, where each call relationship should be confirmed based
936 → on the queried code.
937
938 #### Format Description
939 Your response needs to be provided in JSON format. The specific format for the two choices
940 → is described as follows:
941
942 ##### Format for Returning Input Sequence
943 If you believe that you can find the input byte sequence without further queries, please
944 → write a Python function named generate_buf. When this function is called, it will
945 → randomly generate a byte sequence that meets the requirements.
946 You can call external Python packages within this function to perform specific numerical
947 → calculations. Please include the code for this function and any relevant auxiliary
948 → functions in the code field of the response JSON, and place the call chain from
949 → LLVMFuzzerTestOneInput to the target function in the call_chain field.
950 For example:
951
952

```
```json
953 {
954 "code": "import random\ndef generate_buf():\n frag_num = 1\n frag_total = 1\n session_id = random.randint(0, 99999)\n payload = b'payload'\n data_part =\n f\"AAAAA,{frag_num},{frag_total},{session_id},{payload}\"\\n\n full_sentence =\n f\"!{(data_part)*}\\u0007\"\\n\n return full_sentence.encode('ascii')",
955 "call_chain": ["LLVMFuzzerTestOneInput", "func1", "func2", "target_func"]
956 }
```


957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

```

Listing 5: Tool specification in agent-based setting

```

953
954 ##### Format for Calling the Code Browser
955 The code browser supports the following functions:
956
957 (1). Provide the function name to query the code of that function.
958 (2). Provide the function name to query all the code that calls that function.
959 (3). Provide the global variable name to query all the code that accesses that global
960 → variable.
961 (4). Provide the class/struct member variable name to query all the code that accesses that
962 → class/struct member variable.
963 (5). Provide the global variable name to query the definition code of that global variable.
964 (6). Provide the class/struct/enum type name to query the definition code of that
965 → class/struct/enum type.
966
967 When calling the code browser, the response JSON needs to include the following fields:
968
969 (1). func_name: A list including all the function names whose code needs to be queried; if
970 → there are none, it should be an empty list.
971 (2). called_name: A list including all the function names whose calling code needs to be
972 → queried; if there are none, it should be an empty list.
973 (3). global_used: A list including all the global variable names whose accessing code needs
974 → to be queried; if there are none, it should be an empty list.
975 (4). class_member_used: A list including all the class/struct member variable names whose
976 → accessing code needs to be queried; if there are none, it should be an empty list.
977 (5). global_def: A list including all the global variable names whose definition code need
978 → to be queried; if there are none, it should be an empty list.
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

972
973 (6). type_name: A list including all the class/struct/enum type names whose definitions need
974 → to be queried; if there are none, it should be an empty list.
975
976 For example, the following response:
977
978 ```json
979 {
980     "func_name": ["f", "g"],
981     "called_name": ["h"],
982     "global_used": ["v1"],
983     "class_member_used": ["m1"],
984     "global_def": ["v2"],
985     "type_name": ["clsA"]
986 }
987
988
989 indicates a request to query:
990 + The code for functions f and g
991 + All code that calls function h
992 + All code that accesses the global variable v1
993 + All code that accesses the class/struct member variable m1
994 + The definition code of global variable v2
995 + The definition code of class/structure/enum type clsA
996
997 The user will return the results of the query to you.
998 Do not repeat your query.
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
```

Listing 6: The exemplary interaction in system prompt

```

1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
```

```

1026
1027 By analyzing the code, the variable ch comes from the first element of the data array, and
1028 → we must ensure that size > 0.
1029 I cannot determine how the data array and size are obtained from the input, so I need to
1030 → query which functions called handle_file.
1031
1032 ```json
1033 {
1034     "called_name": ["handle_file"]
1035 }
1036 ```
1037
1038 User:
1039 Here is the code snippet you requested:
1040 ```c
1041 int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
1042       handle_file(data, size);
1043 }
1044 ```
1045
1046 Assistant:
1047 By analyzing the code, the current function call chain to reach the target code is:
1048 LLVMFuzzerTestOneInput → handle_file → func.
1049 In this chain:
1050 LLVMFuzzerTestOneInput calls handle_file via
1051 ```c
1052       handle_file(data, size);
1053 ```
1054
1055 handle_file then calls func via
1056 ```c
1057     if (size > 0){
1058         return func(data[0]);
1059   }
1060 ```
1061
1062 In func, it reaches the target line through
1063 ```c
1064     if (ch == 'A') {
1065         return 1;
1066   }
1067 ```
1068
1069 and all the if conditions must pass.
1070
1071 From the analysis, the variable ch in the func function comes from the first element of the
1072 → data in the handle_file function. The data and size in handle_file are directly passed
1073 → from LLVMFuzzerTestOneInput.
1074 Thus, I can conclude that the constraints to reach the target code are:
1075 The length of the input byte sequence must be > 0.
1076 The first element of the input byte sequence must be 'A'.
1077 The input string b'A' meets these requirements.
1078 Therefore, the final answer is:
1079
```json
{
 "code": "def generate_buf():\n return b'A'\n",
 "call_chain": ["LLVMFuzzerTestOneInput", "handle_file", "func"]
}
```

```

User Prompt. The user prompt is shown in Listing A.3.2, where “code” refers to the 5 lines surrounding the target location, “surrounding_code” is the 10 lines surrounding the target location, and “func_code” is the code of the function queried using the function name.

Listing 7: User prompt in agent-based setting

The point I need to reach is the first line of the following code:

```

```c
{code}
```

```

It is only necessary to execute up to the first line of this code, without considering its
→ content and execution context.

```

1080
1081 To facilitate locating this code, below is a snippet of the surrounding code:
1082 ```c
1083 {surround_code}
1084 ```
1085
1086 This code is located in the function {func_name}, and code of this function is as follows:
1087 ```c
1088 {func_code}
1089 ```
1090 This function is in C/C++ package {prog_package}.
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

```

Query Result Prompt. The template used for responding to LLM’s queries is shown in Listing A.3.2, in which “code_ret” refers to the code snippets returned by the code browser.

Listing 8: Query result prompt in agent-based setting

```

1102
1103
1104 Here is the code snippet you requested:
1105 ```c
1106 {code_ret}
1107 ```
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133

```

B AN EXEMPLARY CONVERSATION WITH LLM

In this section, we present the responses from LLMs that correctly solve the target problems. These examples demonstrate how the LLMs utilize the provided tools and reasoning capabilities to address the challenges posed by the TTG tasks effectively. Each response showcases the model’s understanding of the code, the correct application of reasoning, and the ability to retrieve and integrate relevant information to reach a solution.

B.1 TARGET PROBLEM

The target code location is at the final branch in the starting part of function `png_handle_sCAL`, where the starting part of this function is shown in Listing B.1.

Listing 9: Starting part of target function

```

1112
1113
1114
1115 void /* PRIVATE */
1116 png_handle_sCAL(png_structp png_ptr, png_inforp info_ptr, png_uint_32 length)
1117 {
1118     png_bytеп buffer;
1119     size_t i;
1120     int state;
1121
1122     png_debug(1, "in png_handle_sCAL");
1123
1124     if ((png_ptr->mode & PNG_HAVE_IHDR) == 0)
1125         png_chunk_error(png_ptr, "missing IHDR");
1126
1127     else if ((png_ptr->mode & PNG_HAVE_IDAT) != 0)
1128     {
1129         png_crc_finish(png_ptr, length);
1130         png_chunk_benign_error(png_ptr, "out of place");
1131         return;
1132     }
1133
1134     else if (info_ptr != NULL && (info_ptr->valid & PNG_INFO_sCAL) != 0)
1135     {
1136         png_crc_finish(png_ptr, length);
1137         png_chunk_benign_error(png_ptr, "duplicate");
1138         return;
1139     }
1140     ...
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3099
3100
3101
310
```

1134 B.2 RESPONSE IN RETRIEVAL-BASED SETTING
11351136 The response of the LLM in the retrieval-based setting is shown in Listing B.2.
11371138 Listing 10: Response of LLM in retrieval-based setting
1139

```

# Think:
Okay, let's tackle this problem. The user wants to generate an input byte sequence that will
→ cause the libpng library to execute the specified code block in the function handling
→ the sCAL chunk. The target code is in the else if condition where the info_ptr's valid
→ flag has PNG_INFO_sCAL set, leading to the lines png_crc_finish and the error message
→ "duplicate".
First, I need to figure out how the sCAL chunk is processed in libpng. Looking at the
→ provided code, the function png_handle_sCAL is responsible for handling sCAL chunks. The
→ specific code block they want to hit is when the info_ptr already has the sCAL chunk
→ processed (PNG_INFO_sCAL bit set), which triggers the duplicate error.
So, the conditions to reach that branch are:
1. The info_ptr is not NULL.
2. The info_ptr's valid field has the PNG_INFO_sCAL bit set.
3. The current chunk being processed is another sCAL chunk.

Therefore, to trigger the duplicate error, the input must have two sCAL chunks. The first
→ one will set the valid bit, and the second one will hit the duplicate condition.

Now, how to structure the PNG data to include two sCAL chunks. Let's recall the PNG file
→ structure. A PNG starts with the signature, followed by chunks. Each chunk has a length,
→ type, data, and CRC.

The steps to create such a file:
- Start with the PNG signature (8 bytes: 0x89PNG\r\n\x1a\x0a).
- Include an IHDR chunk (mandatory first chunk).
- Add the first sCAL chunk.
- Add a second sCAL chunk.
- Possibly add an IEND chunk at the end.

The sCAL chunk structure: The chunk type is 'sCAL'. The data for sCAL must start with a unit
→ type (1 byte, either 1 or 2), followed by two null-terminated strings representing the
→ width and height. The minimum length is 4 bytes (unit + at least one byte for each
→ string, but they need to be null-terminated). For example, the first sCAL chunk could
→ have unit 1 followed by "1\02\0" (though exact data might vary as long as it meets the
→ length and format requirements).

Wait, the code for png_handle_sCAL has a check: "else if (info_ptr != NULL &&
→ (info_ptr->valid & PNG_INFO_sCAL) != 0)". So, after the first sCAL chunk is processed,
→ the valid bit is set. The second sCAL chunk will trigger the duplicate check.

Therefore, the input data must have two sCAL chunks. The first one must be valid enough to
→ pass the initial checks (like length >=4, etc.) so that it sets the valid bit. The
→ second one can be minimal, as long as it's recognized as an sCAL chunk again.

Now, constructing the input:

First, the PNG header:
- Signature: 89 50 4E 47 0D 0A 1A 0A

Then, the IHDR chunk:
- Length: 13 (since IHDR data is 13 bytes)
- Type: 73 68 61 6E (IHDR is 'IHDR', so hex 49 48 44 52)
- Data: 00 00 00 01 00 00 01 08 03 (example for 1x1 image, 8 bits, palette)
- CRC: computed for the IHDR data.

Then the first sCAL chunk:
- Length: Let's say 5 bytes (unit type (1) + "1\02\0" would be 1 + 1 + 1 +1 = 4? Wait, the
→ minimum required is 4 bytes (unit + at least 1 byte each for width and height, but they
→ need to be null-terminated. So "1\0" for width and "\02\0" might not be right. Let me
→ think again. The minimum length is 4 bytes as per the code: "else if (length < 4) { ...
→ invalid }". So unit (1 byte) plus the rest must be at least 3 bytes. For example,
→ unit=1, then "1\02\0" would be 4 bytes. So length is 4.

```

```

1188
1189 So first sCAL chunk:
1190 - Length: 4 (in big-endian 00 00 00 04)
1191 - Type: 'sCAL' (hex 73 43 41 4C)
1192 - Data: 01 31 00 32 (unit=1, width "1", height "2" as strings, but need null terminators.
1193 → Wait, the width and height are ASCII numbers separated by a null. So the data would be
1194 → [unit][width string][0][height string][0]. So to have both strings, the minimum would be
1195 → unit (1) + "a\0b\0" which is 4 bytes total. So for example, unit 1, then "1\02\0" would
1196 → be 4 bytes (0x01 0x31 0x00 0x32). That gives unit=1, width "1", height "2". So that's
1197 → acceptable.
1198
1199 CRC for the sCAL data (the 4 bytes) would need to be calculated. But since the input is
1200 → generated, maybe the CRC can be faked, but the library will process it. Wait, but the
1201 → CRC must be correct for the data and length. Hmm, this complicates things because
1202 → generating the correct CRC is necessary for the chunk to be processed. Alternatively,
1203 → perhaps the code allows some flexibility, but the user's input must have valid CRCs for
1204 → each chunk to reach the desired path.
1205
1206 Alternatively, maybe the fuzzer can handle incorrect CRCs but in this case, since the code
1207 → calls png_crc_finish, which verifies the CRC, an incorrect CRC would lead to an error
1208 → before the duplicate check. Therefore, to trigger the duplicate check, the first sCAL
1209 → chunk must have a valid CRC.
1210
1211 So I need to compute the CRC for the first sCAL chunk's data and then append the CRC.
1212
1213 So the first sCAL chunk:
1214
1215 Data part: 0x01 0x31 0x00 0x32 (unit 1, "1\02")
1216
1217 The CRC is computed over the data plus the chunk type. Wait, the CRC in a PNG chunk is
1218 → computed over the chunk type and the data. So for the sCAL chunk:
1219
1220 The CRC is computed as:
1221
1222 CRC32 of the 4-byte chunk type (sCAL) plus the data (4 bytes), then the 4-byte CRC is
1223 → appended.
1224
1225 Wait, the chunk structure is:
1226
1227 [Length (4 bytes)][Chunk Type (4 bytes)][Data (length bytes)][CRC (4 bytes)]
1228
1229 The CRC is computed over the chunk type and data, then stored in the 4 bytes after data.
1230
1231 Therefore, to compute the CRC for the first sCAL chunk:
1232
1233 Data (including the 4 bytes of data):
1234
1235 The chunk type is 'sCAL' (73 43 41 4C in hex, or 0x73 0x43 0x41 0x4C).
1236
1237 The data is 0x01 0x31 0x00 0x32 (assuming the unit is 1, followed by "1\02").
1238
1239 So the bytes to compute the CRC are the 4 bytes of chunk type plus the 4 bytes of data,
1240 → totaling 8 bytes.
1241
1242 Compute the CRC32 of those 8 bytes, then append as 4 bytes.
1243
1244 Then, the second sCAL chunk must be another sCAL chunk. Let's make it minimal. The second
1245 → sCAL chunk can have the same data, or just the minimal required.
1246
1247 Second sCAL chunk:
1248
1249 Length: 4 again (same data as first, or even shorter? Wait, the minimum length is 4, so it
1250 → can be same as first. But to make sure it's processed, it must have a valid CRC.
1251
1252 So the second sCAL chunk would have the same structure as the first, but with the same data,
1253 → leading to same CRC?
1254
1255 Wait, but the data can be different. Let's proceed.
1256
1257 Alternatively, perhaps the second sCAL can have a different data, but must have valid CRC.
1258
1259 So, putting this all together, the input data would be:
1260

```

```

1242
1243
1244     PNG signature (8 bytes) +
1245
1246     IHDR chunk (with length 13, type IHDR, data for 1x1 image, and correct CRC) +
1247
1248     First sCAL chunk (length 4, type sCAL, data as above, correct CRC) +
1249
1250     Second sCAL chunk (same structure as first) +
1251
1252     IEND chunk (mandatory at end, with correct CRC).

1253     Now, the problem is generating all these chunks with correct CRCs. But since the user is to
1254     ↪ write a Python function to generate this, perhaps they can compute the CRCs on the fly.
1255
1256     Wait, but in the code example given in the problem's example, the user didn't compute CRCs
1257     ↪ but just generated the bytes. However, in reality, the CRCs must be correct for the
1258     ↪ chunks to be processed.

1259     Alternatively, maybe the fuzzer's code can handle the CRC, but the input must have the
1260     ↪ correct structure. Since the user's code is to generate the input bytes, perhaps the
1261     ↪ helper functions can compute CRC32.

1262     In Python, the zlib module has a crc32 function. So, in the generate_buf function, we can
1263     ↪ compute the CRCs.

1264     Let me outline the steps for the Python code:
1265
1266     1. Generate the PNG signature.
1267
1268     2. Generate the IHDR chunk:
1269
1270         - IHDR data: For a minimal 1x1 image, 8 bits, palette color type (4). The IHDR data is 13
1271         ↪ bytes: width (4 bytes, 0x00 00 00 01), height (4 bytes same), bit depth (1 byte 08),
1272         ↪ color type (1 byte 03 or 04?), compression (0), filter (0), interlace (0). Wait,
1273         ↪ palette is color type 3. Let's pick 3 for color type.

1274         So IHDR data is:
1275
1276             width: 0x00 00 00 01 (1 pixel)
1277
1278             height: same
1279
1280             bit depth: 0x08 (8 bits)
1281
1282             color type: 0x03 (palette)
1283
1284             compression: 0x00 (deflate)
1285
1286             filter: 0x00 (no filter)
1287
1288             interlace: 0x00 (none)

1289             So data is: b'\x00\x00\x00\x01\x00\x00\x00\x01\x08\x03\x00\x00\x00'

1290             The length of the IHDR data is 13, so the length field is 0x00 00 00 0D.

1291             The CRC is computed over the IHDR type and data. The type is 'IHDR' (0x49 48 44 52).

1292             So the CRC is computed on the 4 bytes of 'IHDR' plus the 13 data bytes.

1293             Using zlib.crc32:
1294
1295                 ihdr_data = b'IHDR' + b'\x00\x00\x00\x01\x00\x00\x00\x01\x08\x03\x00\x00\x00'
1296
1297                 crc = zlib.crc32(ihdr_data) & 0xFFFFFFFF

1298             The CRC for IHDR is known to be 0xed c3 5e 9a (but let's compute it properly in code).

1299
1300     3. First sCAL chunk:
1301
1302         The sCAL chunk data is [unit][width][0][height][0].

```

```

1296
1297
1298     Let's choose unit 1 (0x01), width "1" (0x31), height "2" (0x32), so the data is
1299     → b'\x01\x31\x00\x32'.
1300
1301     The length is 4, so length bytes are 0x00 00 00 04.
1302
1303     The type is 'sCAL' (0x73 43 41 4C).
1304
1305     The CRC is computed over 'sCAL' + data (4 bytes), so total 8 bytes.
1306
1307     So the chunk's CRC is computed on b'sCAL' + b'\x01\x31\x00\x32'.
1308
1309     4. Second sCAL chunk: same as first, so same data and CRC.
1310
1311     5. IEND chunk: standard, with no data. Its length is 0, type IEND. CRC is 0xAE426082.
1312
1313     Putting this together in code:
1314
1315     The Python function generate_buf() would construct all these chunks, compute the CRCs, and
1316     → return the concatenated bytes.
1317
1318     Wait, but the user's code may not need to handle all the CRCs correctly, but the problem
1319     → states that the input must reach the target code, so the first sCAL must be processed
1320     → and set the valid bit, so the CRC must be correct.
1321
1322     Thus, the code must correctly compute CRCs.
1323
1324     So here's the plan for the code:
1325
1326     Import struct and zlib.
1327
1328     Construct each chunk step by step.
1329
1330     First the signature:
1331
1332     signature = b'\x89PNG\r\n\x1a\n'
1333
1334     Then the IHDR chunk:
1335
1336     def make_chunk(type, data):
1337         length = len(data)
1338         crc = zlib.crc32(type.encode() + data) & 0xFFFFFFFF
1339         return struct.pack('!I', length) + type.encode() + data + struct.pack('!I', crc)
1340
1341     Then, create the IHDR data:
1342
1343     ihdr_data = b'\x00\x00\x00\x01' * 2 + b'\x08\x03\x00\x00' # width, height, etc.
1344     Wait, let me write it properly:
1345
1346     width = 1
1347     height = 1
1348     bit_depth = 8
1349     color_type = 3 # palette
1350     compression = 0
1351     filter_method = 0
1352     interlace = 0
1353
1354     ihdr_data = (
1355         width.to_bytes(4, 'big') +
1356         height.to_bytes(4, 'big') +
1357         bytes([bit_depth, color_type, compression, filter_method, interlace]))
1358
1359     # So the data is 13 bytes.
1360
1361     Then the IHDR chunk is made with type 'IHDR' and this data.
1362
1363     Then first sCAL chunk:
1364
1365     scal_data = b'\x01\x31\x00\x32' # unit 1, "1\02"
1366     scal_chunk = make_chunk('sCAL', scal_data)
1367
1368
1369

```

```

1350
1351     Second sCAL chunk: same as first.
1352
1353     Then IEND chunk:
1354
1355     iend_chunk = make_chunk('IEND', b'')
1356
1357     So putting it all together:
1358
1359     def generate_buf():
1360         import struct
1361         import zlib
1362
1363         def make_chunk(type_str, data):
1364             length = len(data)
1365             chunk_type = type_str.encode()
1366             crc_data = chunk_type + data
1367             crc = zlib.crc32(crc_data) & 0xFFFFFFFF
1368             return struct.pack('!I', length) + chunk_type + data + struct.pack('!I', crc)
1369
1370         # Signature
1371         buf = b'\x89PNG\r\n\x1a\n'
1372
1373         # IHDR chunk
1374         ihdr_data = (
1375             (1).to_bytes(4, 'big') * 2 +  # width and height
1376             (8).to_bytes(1, 'big') +      # bit depth
1377             (3).to_bytes(1, 'big') +      # color type (palette)
1378             b'\x00\x00\x00'              # compression, filter, interlace
1379         )
1380         buf += make_chunk('IHDR', ihdr_data)
1381
1382         # First sCAL chunk
1383         scal_data = b'\x01\x31\x00\x32'  # unit 1, width "1", height "2"
1384         buf += make_chunk('sCAL', scal_data)
1385
1386         # Second sCAL chunk (same as first)
1387         buf += make_chunk('sCAL', scal_data)
1388
1389         # IEND chunk
1390         buf += make_chunk('IEND', b'')
1391
1392     return buf
1393
1394     Wait, but need to ensure that after the first sCAL chunk is processed, the info_ptr's valid
1395     ↪ bit is set. The second sCAL chunk will trigger the duplicate check.
1396
1397     This should work. The code creates two sCAL chunks, which will cause the second one to hit
1398     ↪ the duplicate condition.
1399
1400     Now, the call chain from LLVMFuzzerTestOneInput to the target code:
1401
1402     The LLVMFuzzerTestOneInput function is part of the fuzzer's driver, which calls into
1403     ↪ libpng's reading functions. The main path would be:
1404
1405     LLVMFuzzerTestOneInput calls into the libpng read functions, which process each chunk. The
1406     ↪ sCAL chunks are processed via png_handle_sCAL, which is called when the chunk type is
1407     ↪ 'sCAL'.
1408
1409     Thus, the call chain would be something like:
1410
1411     LLVMFuzzerTestOneInput -> png_read_png (or similar) -> processes each chunk -> when 'sCAL'
1412     ↪ chunk is found, calls png_handle_sCAL -> first sCAL sets valid bit, second sCAL triggers
1413     ↪ the duplicate check.
1414
1415     Therefore, the call chain can be listed as:
1416
1417     ["LLVMFuzzerTestOneInput", "png_read_png", "png_process_chunk", "png_handle_sCAL"]
1418
1419     But I need to check the code provided. Looking at the code given, the function handling the
1420     ↪ chunks is part of the png_read functions. The code has functions like png_handle_sCAL,
1421     ↪ which is called when the chunk type is 'sCAL'.
1422
1423

```

1404
 1405
 1406 In the provided code, the function `png_read_data` is part of the read process, but the actual
 1407 handling of each chunk type is via a switch or some dispatch. Looking at the code,
 1408 there's a section where chunks are processed:
 1409
 1410 Looking at the code, there's a function `png_read_chunk` that reads the chunk type and
 1411 dispatches to the appropriate handler. For example, when the chunk type is 'sCAL', it
 1412 calls `png_handle_sCAL`.
 1413
 1414 Thus, the call chain from the fuzzer's entry point (`LLVMFuzzerTestOneInput`) would be:
 1415
 1416 `LLVMFuzzerTestOneInput -> (libpng's read functions) -> png_read_chunk -> png_handle_sCAL`
 1417
 1418 So the call chain would be:
 1419
 1420 `["LLVMFuzzerTestOneInput", "png_read_png", "png_read_chunk", "png_handle_sCAL"]`
 1421
 1422 But the exact names might be different. Since the code provided includes the functions like
 1423 `png_handle_sCAL`, the call chain is:
 1424
 1425 `LLVMFuzzerTestOneInput calls into the libpng processing, which reads the first sCAL chunk`
 1426 `& calls png_handle_sCAL, which sets the valid bit. Then, when the second sCAL chunk is`
 1427 `& read, the same function is called again, leading to the duplicate check.`
 1428
 1429 Thus, the `call_chain` field should list the path from the fuzzer entry to the target code:
 1430
 1431 `["LLVMFuzzerTestOneInput", "png_process_chunks", "png_handle_sCAL"]`
 1432
 1433 But to be precise, based on the provided code, the code that processes each chunk after
 1434 reading the chunk type is likely in a function like `process_chunk`, which calls the
 1435 respective handler. Since the code provided includes the function `png_handle_sCAL`, the
 1436 call chain would be:
 1437
 1438 `LLVMFuzzerTestOneInput -> (the code that reads the PNG data) -> when a 'sCAL' chunk is`
 1439 `& encountered, calls png_handle_sCAL. The first call sets the valid bit, the second call`
 1440 `& triggers the target code.`
 1441
 1442 Therefore, the call chain to the target line (the duplicate check in `png_handle_sCAL`) is:
 1443
 1444 `LLVMFuzzerTestOneInput -> (the libpng read loop) -> png_handle_sCAL (second time).`
 1445
 1446 So the `call_chain` is `["LLVMFuzzerTestOneInput", "png_handle_sCAL"]`
 1447
 1448 Wait, but that's too short. The actual call path would go through more functions. Let me
 1449 think again.
 1450
 1451 The fuzzer's function is the entry point, then the libpng library's functions are called to
 1452 read the PNG data. The code for the fuzzer's `LLVMFuzzerTestOneInput` function is provided
 1453 as:
 1454
 1455

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
  1456     if (size < kPngHeaderSize) {
  1457         return 0;
  1458     }
  1459
  1460     ...
  1461     }
  1462
  1463     png_read_end(png_handler.png_ptr, png_handler.end_info_ptr);
  1464
  1465     PNG_CLEANUP
  1466     return 0;
  1467 }
```


 1468
 1469 The code likely initializes a `png_ptr` and reads the data via `png_read_png` or similar. The
 1470 actual processing of each chunk is done in the `png_read` functions. The chunk processing
 1471 is handled by functions like `png_read_chunk` and then the respective handlers (like
 1472 `png_handle_sCAL`).
 1473

```

1458
1459 Thus, the call chain is:
1460 LLVMFuzzerTestOneInput -> (initialization and calls to libpng's read functions) -> when the
1461   ↪ second sCAL chunk is processed, it calls png_handle_sCAL again, leading to the target
1462   ↪ code.
1463 Therefore, the call_chain should list the path from the entry point to the target function.
1464   ↪ The target function is png_handle_sCAL, but specifically the part where the duplicate
1465   ↪ check is made. So the chain is:
1466
1467 LLVMFuzzerTestOneInput -> ... -> png_handle_sCAL (the second time it is called)
1468
1469 But the call chain is the path taken to reach the target line, which is inside
1470   ↪ png_handle_sCAL. So the call chain from the entry point to the function containing the
1471   ↪ target code is:
1472
1473 ["LLVMFuzzerTestOneInput", "png_read_png", "png_process_chunks", "png_handle_sCAL"]
1474
1475 Alternatively, if the code structure is that the main loop in libpng reads chunks and
1476   ↪ dispatches to handlers, then the chain is:
1477 LLVMFuzzerTestOneInput -> png_read_start -> ... -> png_read_chunk -> png_handle_sCAL.
1478
1479 But since the exact code isn't fully provided, I'll assume the minimal chain that includes
1480   ↪ the handler function.
1481 The key point is that the code in question is in png_handle_sCAL, so the call chain must
1482   ↪ include that function.
1483 Thus, the final answer includes the Python code generating the two sCAL chunks with correct
1484   ↪ CRCs, and the call_chain as listed above.
1485
1486 Wait, in the code provided, the function png_handle_sCAL has the code block:
1487
1488 else if (info_ptr != NULL && (info_ptr->valid & PNG_INFO_sCAL) != 0)
1489 {
1490     png_crc_finish(png_ptr, length);
1491     png_chunk_benign_error(png_ptr, "duplicate");
1492     return;
1493 }
1494
1495 So to reach this, the first sCAL chunk must have been processed, setting the valid bit, and
1496   ↪ the second sCAL chunk's processing will hit this condition.
1497 The code for generate_buf must create two sCAL chunks. The first one must be valid (so that
1498   ↪ it processes and sets the valid bit). The second one can be minimal but must be
1499   ↪ recognized as an sCAL chunk.
1500 Therefore, the Python function should generate the structure with two sCAL chunks.
1501
1502 Now, testing the code:
1503 The first sCAL's data is 4 bytes (unit + "1\02"), which meets the minimum length of 4 bytes.
1504   ↪ The second sCAL can be the same.
1505 Thus, the code should work.
1506
1507 # Answer:
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3199
3199
3200
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3379
3380
3381
3382
3383
3384
3385
3386
3387
3388
3389
3389
3390
3391
3392
3393
3394
3395
3396
3397
3398
3398
3399
3399
3400
3401
3402
3403
3404
3405
3406
3407
3408
3409
3409
3410
3411
3412
3413
3414
3415
3416
3417
3418
3419
3419
3420
3421
3422
3423
3424
3425
3426
3427
3428
3429
3429
3430
3431
3432
3433
3434
3435
3436
3437
3438
3439
3439
3440
3441
3442
3443
3444
3445
3446
3447
3448
3449
3449
3450
3451
3452
3453
3454
3455
3456
3457
3458
3459
3459
3460
3461
3462
3463
3464
3465
3466
3467
3468
3469
3469
3470
3471
3472
3473
3474
3475
3476
3477
3478
3479
3479
3480
3481
3482
3483
3484
3485
3486
3487
3488
3489
3489
3490
3491
3492
3493
3494
3495
3496
3497
3498
3499
3499
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3509
3510
3511
3512
3513
3514
3515
3516
3517
3518
3519
3519
3520
3521
3522
3523
3524
3525
3526
3527
3528
3529
3529
3530
3531
3532
3533
3534
3535
3536
3537
3538
3539
3539
3540
3541
3542
3543
3544
3545
3546
3547
3548
35
```

1512 B.3 RESPONSE IN AGENT-BASED SETTING
1513

1514

1515 The response of the LLM in the first round is shown in Listing B.3, and the second response is
1516 shown in Listing B.3.

1517

1518

1519 Listing 11: Response of LLM in the first round of agent-based setting
1520

1521

```

# Think:
Okay, I need to generate an input byte sequence that will reach the specific line in the
→ png_handle_sCAL function where png_crc_finish is called with the "duplicate" error. Let
→ me start by understanding the code structure of png_handle_sCAL.
The target code is inside the else if condition: else if (info_ptr != NULL &&
→ (info_ptr->valid & PNG_INFO_sCAL) != 0). So to trigger this, info_ptr must not be NULL,
→ and the PNG_INFO_sCAL flag must be already set in info_ptr->valid.
Now, the function png_handle_sCAL is part of libpng, which processes PNG chunks. The sCAL
→ chunk is a special chunk, so the input must be a PNG file containing an sCAL chunk. But
→ the problem is to get the code to process this chunk and hit the duplicate case.
First, I need to trace the call chain from LLVMFuzzerTestOneInput to png_handle_sCAL. Let's
→ start by finding out how the fuzzer input is processed. Typically, in libpng fuzzers,
→ the input is treated as a PNG file. The entry point LLVMFuzzerTestOneInput probably
→ calls libpng functions to read and parse the PNG data.

So, the first step is to check how LLVMFuzzerTestOneInput is implemented. The user hasn't
→ provided that code yet. Without knowing how the input is passed to libpng's parsing
→ functions, I can't determine the path. Therefore, I need to request the code for
→ LLVMFuzzerTestOneInput.

Once I have that, I can see how the data is processed. Suppose LLVMFuzzerTestOneInput calls
→ png_read functions, which eventually process chunks. The sCAL chunk handler is
→ png_handle_sCAL. So, the fuzzer input must include an sCAL chunk.

But even if the input has an sCAL chunk, the code will reach png_handle_sCAL. However, the
→ target condition is that info_ptr's valid field already has PNG_INFO_sCAL set. That
→ implies that the info struct must have already processed an sCAL chunk before. So,
→ perhaps the input needs to contain two sCAL chunks. The first one would set the
→ PNG_INFO_sCAL flag, and the second one would trigger the duplicate check.

Wait, but how does libpng process multiple chunks? When a PNG file is parsed, each chunk is
→ processed in order. If the first sCAL chunk is processed, then info_ptr's valid bit is
→ set. Then, if a second sCAL chunk is encountered, the code would check if the valid bit
→ is already set, leading to the duplicate error.

So, the input needs to be a PNG file with two sCAL chunks. However, the fuzzer's input is a
→ byte stream. The LLVMFuzzerTestOneInput function likely passes this data to libpng's
→ parsing functions. Therefore, the fuzzer's input must be a valid PNG file that includes
→ two sCAL chunks.

But to construct such a PNG file, I need to know the structure. PNG files start with a
→ signature, followed by a series of chunks. Each chunk has a length, type, data, and CRC.
→ So, to create two sCAL chunks, I need to structure the input correctly.

But let's confirm the call chain. The user's target is in png_handle_sCAL, which is called
→ when the sCAL chunk is processed. So, the input must include an sCAL chunk. The code
→ path to reach the duplicate error requires that when the sCAL chunk is processed, the
→ info_ptr already has PNG_INFO_sCAL set.

So, perhaps the input must have two sCAL chunks. The first one is processed normally,
→ setting the valid flag. The second one triggers the duplicate error.

Therefore, the steps would be:

1. The input starts with the PNG signature.
2. IHDR chunk (required).
3. First sCAL chunk.
4. Second sCAL chunk.
5. Other necessary chunks (like IDAT, IEND).

```

```

1566
1567 But wait, the code in png_handle_sCAL checks if the mode has PNG_HAVE_IDAT. If so, it skips
1568 → processing. So the sCAL chunk must come before the IDAT chunk. In PNG structure, chunks
1569 → after IDAT are not allowed except for IEND. So the sCAL chunks must be between IHDR and
1570 → IDAT.
1571
1572 Moreover, the code checks if (png_ptr->mode & PNG_HAVE_IDAT) != 0, which means if IDAT has
1573 → been processed, the sCAL chunk is out of place. So the sCAL chunks must be before IDAT.
1574
1575 Therefore, the input must be structured as:
1576
1577 PNG signature → IHDR → sCAL → sCAL → IDAT → IEND.
1578
1579 But creating such a PNG file requires precise byte crafting. However, the fuzzer's input is
1580 → passed via LLVMFuzzerTestOneInput. So the code for that function must be parsing the
1581 → input as a PNG file. Let's check how the fuzzer test function is structured.
1582
1583 But since the user hasn't provided the code for LLVMFuzzerTestOneInput, I can't proceed.
1584 → Therefore, my first step is to query the code of LLVMFuzzerTestOneInput to see how data
1585 → is processed. Additionally, I need to confirm how the libpng functions are called in the
1586 → fuzzer. For example, the fuzzer might initialize png_struct and png_info structs, then
1587 → read the PNG data.
1588
1589 Once I have the code for LLVMFuzzerTestOneInput, I can see how the data is passed to libpng.
1590 → Let's assume that the function uses png_read functions to parse the input data. Then,
1591 → the input needs to be a valid PNG file with two sCAL chunks.
1592
1593 Moreover, the info_ptr must not be NULL. In libpng, info_ptr is created by
1594 → png_create_info_struct. So the fuzzer probably creates it, so info_ptr is not NULL.
1595
1596 So, the key constraints are:
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619

```

The input is a valid PNG file with correct signature.
 → IHDR chunk is present.
 → Two sCAL chunks after IHDR and before IDAT.
 → The first sCAL chunk is valid, so that it sets PNG_INFO_sCAL in info_ptr->valid.
 → The second sCAL chunk triggers the duplicate check.

But to construct the sCAL chunks correctly, each must have valid data. The sCAL chunk's data
 → starts with a unit byte (1 or 2), followed by ASCII width and height separated by a null
 → byte.

So, the first sCAL chunk must be correctly formatted so that it is accepted, setting the
 → valid flag. Then the second sCAL chunk, when processed, will hit the duplicate
 → condition.

Thus, the input bytes need to form a PNG file with these chunks. But generating such a byte
 → sequence requires knowing the exact structure.

Alternatively, maybe the fuzzer input is not required to be a full PNG file. But given that
 → the code checks for PNG_HAVE_IHDR and other mode flags, the input must at least have the
 → necessary chunks to pass those checks.

But without seeing how LLVMFuzzerTestOneInput processes the data, it's impossible to
 → proceed. Therefore, I need to request the code for LLVMFuzzerTestOneInput and any
 → relevant functions in the call chain leading to png_handle_sCAL.

Possible steps for code queries:

1. Query the code of LLVMFuzzerTestOneInput to see how data is processed.
2. Determine which functions are called from there. For example, maybe the fuzzer uses
 → png_read_png or similar functions.
3. Check if info_ptr is created (png_create_info_struct) and passed.
4. Check the call chain to see how the sCAL chunk is processed.

Therefore, the initial JSON query should be to get the code for LLVMFuzzerTestOneInput and
 → any functions it calls that are part of the processing to handle chunks, especially
 → sCAL.

So, the first JSON query would be:

```
{
}
```

```
1620 "func_name": ["LLVMFuzzerTestOneInput"],
1621 "called_name": [],
1622 "global_used": [],
1623 "class_member_used": [],
1624 "global_def": [],
1625 "type_name": []
1626 }
1627
1628 Once I have that, I can see how data is passed into the libpng functions. For example, if
1629 → LLVMFuzzerTestOneInput initializes png_struct and png_info, then starts reading the
1630 → data.
1631
1632 After getting that code, I may need to follow the call chain further. For example, if
1633 → LLVMFuzzerTestOneInput calls a function like png_read_png, which in turn processes
1634 → chunks, leading to png_handle_sCAL.
1635
1636 But to confirm, I might need to query the code of functions called by
1637 → LLVMFuzzerTestOneInput, such as png_read_png, or other functions in the call chain.
1638
1639 Another consideration: the fuzzer might have a custom setup. For example, it might use a
1640 → file reader that's passed to libpng, which reads from the input buffer.
1641
1642 Assuming that the code for LLVMFuzzerTestOneInput is something like:
1643
1644 #include <png.h>
1645
1646 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
1647     png_structp png = png_create_read_struct(...);
1648     png_infop info = png_create_info_struct(png);
1649     // Set up error handling, etc.
1650     png_set_read_fn(png, &data, user_read_fn);
1651     png_read_info(png, info);
1652     // ... process chunks
1653     png_destroy_read_struct(...);
1654     return 0;
1655 }
1656
1657 In this case, the input data is treated as a PNG file. So the input must be a valid PNG file
1658 → with the required chunks.
1659
1660 But to reach png_handle_sCAL, the input must include an sCAL chunk. Moreover, the duplicate
1661 → case requires two sCAL chunks. But in PNG, chunks are processed in order. The first sCAL
1662 → would set the info_ptr's valid flag, the second would trigger the duplicate error.
1663
1664 Therefore, the input must be a PNG file with two sCAL chunks. However, creating such a PNG
1665 → requires precise construction, including valid chunk lengths, types, data, and CRCs.
1666
1667 But generating this manually is complex. However, for fuzzing purposes, perhaps the fuzzer
1668 → can be tricked into processing the sCAL chunk twice without having to generate a valid
1669 → PNG. But given the code checks (like PNG_HAVE_IHDR, etc.), the input must at least have
1670 → the necessary chunks to pass these checks.
1671
1672 Alternatively, maybe the code can be reached by a malformed PNG that still manages to
1673 → trigger the sCAL processing twice.
1674
1675 But without knowing how the fuzzer passes data to libpng, it's hard to proceed. Therefore,
1676 → the immediate step is to query the code of LLVMFuzzerTestOneInput and any functions it
1677 → calls that are part of processing the input data.
1678
1679 Another possibility is that the fuzzer uses a custom IO reader that feeds data to libpng.
1680 → The code for LLVMFuzzerTestOneInput might set up the read function to read from the data
1681 → buffer. But without seeing that code, I can't determine the exact requirements.
1682
1683 So, the next step is to request the code for LLVMFuzzerTestOneInput. Once I have that, I can
1684 → trace the call chain further.
1685
1686 Therefore, the initial query should be:
1687
1688 ```json
1689 {
1690     "func_name": ["LLVMFuzzerTestOneInput"],
1691 }
```

```

1674
1675     "called_name": [],
1676     "global_used": [],
1677     "class_member_used": [],
1678     "global_def": [],
1679     "type_name": []
1680 }
1681 ...
1682 # Answer:
1683 ...
1684 {
1685     "func_name": ["LLVMFuzzerTestOneInput"],
1686     "called_name": [],
1687     "global_used": [],
1688     "class_member_used": [],
1689     "global_def": [],
1690     "type_name": []
1691 }
1692 ...

```

Listing 12: Response of LLM in the second round of agent-based setting

```

1692 # Think:
1693 Okay, let's try to figure out how to generate an input that reaches the target line in
1694 ↪ png_handle_sCAL. The user wants the input to cause the program to execute the line where
1695 ↪ png_crc_finish is called with the "duplicate" error.
1696
1697 First, looking at the provided code for LLVMFuzzerTestOneInput, it processes the input as a
1698 ↪ PNG file. The function checks if the input starts with the PNG header (kPngHeaderSize is
1699 ↪ 8 bytes, the standard PNG signature). If it does, it proceeds to read the PNG using
1700 ↪ libpng functions.
1701
1702 The key steps in LLVMFuzzerTestOneInput are:
1703 1. Check the input size is at least the PNG header size.
1704 2. Verify the PNG signature.
1705 3. Create png_struct and png_info structures.
1706 4. Set up custom read functions to read from the input buffer after the header.
1707 5. Read the PNG info, IHDR chunk, and process the image data.
1708 To reach png_handle_sCAL, the input must include an sCAL chunk. The target code is triggered
1709 ↪ when a second sCAL chunk is encountered after the first one has already set the
1710 ↪ PNG_INFO_sCAL flag in info_ptr.
1711 So, the input PNG needs to have two sCAL chunks. The first sCAL chunk is processed normally,
1712 ↪ setting the valid flag. The second sCAL chunk will trigger the duplicate check.
1713 But how does the PNG structure work? PNG chunks must appear in a certain order. The IHDR
1714 ↪ must come first, followed by other chunks like sCAL, then IDAT, and finally IEND.
1715 ↪ However, the code in png_handle_sCAL checks if IDAT has been processed (PNG_HAVE_IDAT).
1716 ↪ If so, it skips processing the sCAL chunk. Therefore, the sCAL chunks must come before
1717 ↪ IDAT.
1718
1719 So, the structure of the input should be:
1720 - PNG signature (8 bytes)
1721 - IHDR chunk
1722 - sCAL chunk (first)
1723 - sCAL chunk (second)
1724 - IDAT chunk(s)
1725 - IEND chunk
1726
1727 Each chunk has a length (4 bytes), type (4 bytes), data, and CRC (4 bytes). For the sCAL
1728 ↪ chunks, the data must be valid to pass initial checks. The first sCAL chunk needs to be
1729 ↪ correctly formatted so that it's accepted, setting the PNG_INFO_sCAL flag.
1730
1731 Looking at the code for png_handle_sCAL, the sCAL chunk's data must start with a unit byte
1732 ↪ (1 or 2), followed by width and height as ASCII strings separated by a null byte. The
1733 ↪ png_check_fp_number function checks the format of these strings.
1734
1735 To create a valid first sCAL chunk:
1736 - Length: Let's say the data is "1\01.0\01.0" (unit=1, width=1.0, height=1.0). The actual
1737 ↪ length would be 1 (unit) + 3 (width "1.0") + 1 (null) + 3 (height "1.0") = 8 bytes. So
1738 ↪ length is 8, stored as 4 bytes in big-endian.
1739 - Type: 'sCAL' (ASCII, 4 bytes)

```

```

1728
1729   - Data: The unit byte followed by "1.0\01.0"
1730   - CRC: Calculated over type and data.
1731
1732   The second sCAL chunk can be minimal, maybe even invalid, since we just need it to trigger
1733   → the duplicate check. But its type must be 'sCAL', and the length must be at least 4
1734   → (since the code checks length < 4 for some conditions). However, since the code checks
1735   → if (info_ptr->valid & PNG_INFO_sCAL) != 0, the second sCAL chunk just needs to exist.
1736
1737   But the code for the first sCAL chunk must pass all checks to set the valid flag. Let's
1738   → outline the steps for the first sCAL chunk:
1739
1740   1. The chunk's length is >=4 (the code checks else if (length <4) ...).
1741   2. The unit is 1 or 2.
1742   3. The width and height are valid floating-point numbers in ASCII, separated by a null byte.
1743   4. The CRC is correct.
1744
1745   So, the first sCAL chunk must be correctly formatted. The second sCAL chunk just needs to be
1746   → present, but since the code checks the valid flag, its data may not matter as long as
1747   → the chunk is processed.
1748
1749   Constructing such a PNG file requires precise byte crafting. Let's outline the structure:
1750
1751   PNG signature (8 bytes): \x89PNG\r\n\x1a\x0d
1752
1753   IHDR chunk (25 bytes total):
1754   - Length: 13 (00 00 00 0D)
1755   - Type: IHDR (49 48 44 52)
1756   - Data: width (4 bytes), height (4 bytes), bit depth (1), color type (1), compression (1),
1757   → filter (1), interlace (1)
1758   - CRC: computed over type and data.
1759
1760   Then the first sCAL chunk:
1761   - Length: 8 (00 00 00 08)
1762   - Type: sCAL (73 43 41 4C)
1763   - Data: unit (1, 0x01), width "1.0" (31 2E 30), null (00), height "1.0" (31 2E 30)
1764   - CRC: calculated.
1765
1766   Second sCAL chunk:
1767   - Length: 4 (00 00 00 04)
1768   - Type: sCAL (73 43 41 4C)
1769   - Data: unit (1, 0x01) followed by three arbitrary bytes (since length is 4, but the code
1770   → requires more, but perhaps this will trigger the error early)
1771   - CRC: doesn't matter as long as the chunk is processed.
1772
1773   Then IDAT chunk(s) and IEND.
1774
1775   However, the code in png_handle_sCAL for the first sCAL chunk must pass the checks. The
1776   → first sCAL chunk's data must be valid. The code reads the entire chunk data, then checks
1777   → the unit, then the width and height.
1778
1779   For the first sCAL chunk:
1780   - buffer[0] must be 1 or 2.
1781   - The rest of the data must have width and height as valid numbers.
1782
1783   In the code, after reading the data, it checks:
1784   if (buffer[0] != 1 && buffer[0] != 2) → error.
1785   Then, check the numbers:
1786   i starts at 1. png_check_fp_number checks the width. Then there must be a null, then the
1787   → height.
1788
1789   So for the first sCAL chunk, data should be:
1790   unit (1) followed by "1.0\0" and "1.0", totaling 1 + 3 + 1 + 3 = 8 bytes. So length is 8.
1791
1792   The second sCAL chunk can have length 4, but according to the code:
1793   else if (length < 4) → invalid. So length must be >=4. So length 4 is okay, but data must be
1794   → at least 4 bytes. The code will read 4 bytes, then check buffer[0], but since the
1795   → info_ptr already has PNG_INFO_sCAL, it will trigger the duplicate error.
1796
1797   But wait, the code for the duplicate check is:
1798   else if (info_ptr != NULL && (info_ptr->valid & PNG_INFO_sCAL) != 0)
1799
1800
1801

```

1782
 1783 So when processing the second sCAL chunk, since `info_ptr->valid` has `PNG_INFO_sCAL` set, it
 1784 \hookrightarrow will enter this condition, call `png_crc_finish`, and return. So regardless of the data in
 1785 \hookrightarrow the second sCAL chunk, as long as the chunk is processed and the valid flag is set, it
 1786 \hookrightarrow will trigger the target line.

1787 Therefore, the input must have two sCAL chunks. The first must be valid to set the flag, the
 1788 \hookrightarrow second can be minimal but valid enough to be processed (length ≥ 4 , correct CRC).

1789 Now, constructing such a PNG:

1790 1. PNG signature (8 bytes).
 1791 2. IHDR chunk (25 bytes).
 1792 3. First sCAL chunk (8 + 4 (type) + 8 (data) + 4 (CRC) = 24 bytes).
 1793 4. Second sCAL chunk (4 + 4 + 4 + 4 = 16 bytes, with data length 4, but valid CRC).
 1794 5. IDAT chunk (minimal, perhaps 1 byte data).
 1795 6. IEND chunk.

1796 But generating the correct CRC values is essential. The CRC is computed over the chunk type
 1797 \hookrightarrow and data. For the first sCAL chunk, the type is 'sCAL' and the data is as above. For the
 1798 \hookrightarrow second sCAL chunk, the type is 'sCAL' and data is 4 bytes (e.g., 0x01 0x00 0x00 0x00).
 1799 \hookrightarrow However, the CRC must be correct, otherwise libpng will ignore the chunk due to CRC
 \hookrightarrow errors.

1800 But in the code provided, `png_set_crc_action` is called with `PNG_CRC QUIET USE`, which means
 1801 \hookrightarrow it will use the CRC without complaining. Wait, the code has:

1802 `png_set_crc_action(png_handler.png_ptr, PNG_CRC QUIET USE, PNG_CRC QUIET USE);`

1803 According to libpng documentation, `PNG_CRC QUIET USE` means that if the CRC is incorrect, it
 1804 \hookrightarrow will still use the data but not complain. However, if `png_crc_finish` is called, it will
 1805 \hookrightarrow check the CRC. Wait, in `png_handle_sCAL`, after reading the chunk data, it calls
 1806 \hookrightarrow `png_crc_finish(png_ptr, 0)`. This checks the CRC of the chunk. If the CRC is invalid, it
 1807 \hookrightarrow will return early, thus not setting the `PNG_INFO_sCAL` flag.

1808 Therefore, for the first sCAL chunk, the CRC must be correct; otherwise, the code will not
 1809 \hookrightarrow set the valid flag. Similarly, the second sCAL chunk's CRC must be correct to be
 1810 \hookrightarrow processed and trigger the duplicate error.

1811 Calculating CRCs can be complex. However, for the purpose of generating a valid input, we
 1812 \hookrightarrow need to compute the correct CRC for each chunk.

1813 Alternatively, perhaps we can craft the input in such a way that the CRC is correct. This
 1814 \hookrightarrow would require computing the CRC for each chunk.

1815 Let's outline the steps to generate the input:

1816 1. Create the PNG signature: 8 bytes (0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A).
 1817 2. IHDR chunk:
 1818

- Length: 13 (0x00 0x00 0x00 0x0D)
- Type: 'IHDR' (0x49 0x48 0x44 0x52)
- Data: For simplicity, set width=1, height=1, bit_depth=8, color_type=2 (RGB),
 1819 \hookrightarrow compression=0, filter=0, interlace=0.
- CRC: Compute over 'IHDR' + data. Need to calculate this.

 1820 3. First sCAL chunk:
 1821

- Length: 8 (0x00 0x00 0x00 0x08)
- Type: 'sCAL' (0x73 0x43 0x41 0x4C)
- Data: unit=1 (0x01), followed by "1.0\01.0" (hex: 0x31 0x2E 0x30 0x00 0x31 0x2E 0x30)
- CRC: Compute over 'sCAL' + data.

 1822 4. Second sCAL chunk:
 1823

- Length: 4 (0x00 0x00 0x00 0x04)
- Type: 'sCAL' (0x73 0x43 0x41 0x4C)
- Data: unit=1 (0x01) followed by three zeros (0x00 0x00 0x00)
- CRC: Compute over 'sCAL' + data.

 1824 5. IDAT chunk:
 1825

- Minimal data, perhaps one byte. But IDAT requires valid zlib-compressed data. However,
 1826 \hookrightarrow in fuzzing, maybe the code doesn't process the IDAT data deeply. Alternatively, use a
 1827 \hookrightarrow zero-length IDAT chunk.

```

1836
1837
1838 6. IEND chunk:
1839   - Length 0, type 'IEND', CRC.
1840
1841   However, generating valid IDAT data is complicated. But in the provided
1842   ↪ LLVMFuzzerTestOneInput, after reading the info, it processes the image data by reading
1843   ↪ rows. If the image dimensions are small (1x1), and the IDAT data is valid, this might be
1844   ↪ manageable.
1845
1846   Alternatively, perhaps the code can process the IDAT chunk even with invalid data, as the
1847   ↪ fuzzer's error handling may allow it. However, the code checks if png_read_row succeeds.
1848   ↪ If the IDAT data is invalid, the read might fail, but since the target is to reach the
1849   ↪ sCAL handling before that, maybe it's sufficient.
1850
1851   But given the complexity, perhaps the minimal approach is to create a valid PNG with two
1852   ↪ sCAL chunks, correct CRCs, and minimal valid IDAT data.
1853
1854   To generate this programmatically, the Python code can construct the PNG bytes step by step,
1855   ↪ calculating CRCs where necessary.
1856
1857   For CRC calculation, the zlib library in Python has a crc32 function, which can be used.
1858   ↪ Note that PNG uses CRC-32 with the IEEE 802.3 polynomial.
1859
1860   Putting it all together:
1861
1862   - The first sCAL chunk must have valid data and correct CRC.
1863   - The second sCAL chunk must have correct CRC to be processed, but its data can be minimal.
1864
1865   The Python function generate_buf can construct this byte sequence.
1866
1867   The call chain is LLVMFuzzerTestOneInput → libpng's parsing functions (png_read_info, etc.)
1868   ↪ → png_handle_sCAL when processing each sCAL chunk.
1869
1870   So the call chain from LLVMFuzzerTestOneInput to the target function would be
1871   ↪ LLVMFuzzerTestOneInput → png_read_info → ... (libpng internal functions) →
1872   ↪ png_handle_sCAL.
1873
1874
1875   But the user's example in the session used direct function names in the call_chain. Since
1876   ↪ the exact path through libpng may involve internal functions, but the key is that
1877   ↪ LLVMFuzzerTestOneInput calls into libpng which calls png_handle_sCAL. However, the
1878   ↪ user's example in the session included the direct call chain. For the answer, the
1879   ↪ call_chain should list LLVMFuzzerTestOneInput leading to png_handle_sCAL via the
1880   ↪ necessary functions. However, based on the code provided, the call chain would be
1881   ↪ LLVMFuzzerTestOneInput → png_read_info → png_read_info_impl → png_handle_sCAL (assuming
1882   ↪ libpng's internal call hierarchy). But without seeing the code of png_read_info, it's
1883   ↪ hard to know. However, the user's example included a direct call chain, so perhaps the
1884   ↪ call_chain should be LLVMFuzzerTestOneInput → png_read_info → png_handle_sCAL. But since
1885   ↪ the user may not have provided all intermediate functions, perhaps the call_chain is
1886   ↪ written as ["LLVMFuzzerTestOneInput", "png_read_info", "png_handle_sCAL"].
1887   But according to the code provided, LLVMFuzzerTestOneInput calls png_read_info, which in
1888   ↪ libpng processes the chunks. When an sCAL chunk is encountered, png_handle_sCAL is
1889   ↪ called. So the call chain is LLVMFuzzerTestOneInput → png_read_info → (libpng internal
1880   ↪ processing) → png_handle_sCAL. However, for the purposes of the answer, it's acceptable
1881   ↪ to list the direct functions called from LLVMFuzzerTestOneInput leading to the target.
1882   ↪ Since the user's code shows that the fuzzer calls png_read_info, which leads to
1883   ↪ processing chunks including sCAL, the call_chain would be ["LLVMFuzzerTestOneInput",
1884   ↪ "png_read_info", "png_handle_sCAL"].
1885   Thus, the final answer will construct a PNG with two sCAL chunks, ensuring the first is
1886   ↪ valid to set the flag and the second triggers the duplicate error.
1887
1888 # Answer:
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899

```

```

1890
1891 "code": "import zlib\n\ndef generate_buf():\n    # PNG signature\n    png_sig =\n    ↪ bytes([0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A])\n    ↪ \n    # IHDR chunk (1x1\n    ↪ RGB)\n    ↪ ihdr_data = bytes([\n        0x00, 0x00, 0x00, 0x01, # width\n        0x00, 0x00, 0x01, # height\n        0x08, # bit depth\n        0x02, #\n        color type (RGB)\n        0x00, # compression\n        0x00, # filter\n        0x00 # interlace\n    ])\n    ↪ ihdr_type = b'IHDR'\n    ↪ ihdr_crc =\n    ↪ zlib.crc32(ihdr_type + ihdr_data).to_bytes(4, 'big')\n    ↪ ihdr_chunk = bytes([0x00,\n        0x00, 0x00, 0x0D]) + ihdr_type + ihdr_data + ihdr_crc\n    ↪ \n    # First valid sCAL\n    ↪ chunk\n    ↪ scal1_data = bytes([0x01]) + b'1.0'\n    ↪ scal1_type = b'sCAL'\n    ↪ scal1_crc = zlib.crc32(scal1_type + scal1_data).to_bytes(4, 'big')\n    ↪ scal1_chunk = bytes([0x00, 0x00, 0x00, 0x08]) + scal1_type + scal1_data + scal1_crc\n    ↪ \n    # Second sCAL chunk (duplicate)\n    ↪ scal2_data = bytes([0x01, 0x00, 0x00, 0x00])\n    ↪ scal2_type = b'sCAL'\n    ↪ scal2_crc = zlib.crc32(scal2_type +\n        scal2_data).to_bytes(4, 'big')\n    ↪ scal2_chunk = bytes([0x00, 0x00, 0x00, 0x04]) +\n        scal2_type + scal2_data + scal2_crc\n    ↪ \n    # IDAT chunk (minimal valid zlib data\n    ↪ for 1x1 RGB)\n    ↪ idat_data = zlib.compress(b'\\x00\\x00\\x00\\x00')\n    ↪ idat_type = b'IDAT'\n    ↪ idat_crc = zlib.crc32(idat_type + idat_data).to_bytes(4,\n        'big')\n    ↪ idat_chunk = len(idat_data).to_bytes(4, 'big') + idat_type + idat_data +\n        idat_crc\n    ↪ \n    # IEND chunk\n    ↪ iend_chunk = bytes([0x00, 0x00, 0x00, 0x00]) +\n        b'IEND' + zlib.crc32(b'IEND').to_bytes(4, 'big')\n    ↪ \n    return png_sig +\n    ↪ ihdr_chunk + scal1_chunk + scal2_chunk + idat_chunk + iend_chunk",
1907 "call_chain": ["LLVMFuzzerTestOneInput", "png_read_info", "png_handle_sCAL"]
1908 }
1909 ...
1910
1911
1912 C CASE STUDIES
1913
1914 In this section, we perform case studies to understand why LLMs fail to solve TTG problems. We
1915 analyze the failure reasons and select the relevant response snippets during reasoning to illustrate
1916 these issues.
1917
1918 C.1 FAILURE TYPE 1: HALLUCINATION IN REASONING MODELS
1919
1920 By manually analyzing the responses from LLMs, we found that in many cases, reasoning LLMs
1921 tend to hallucinate and rely on fabricated code snippets for decision-making rather than querying the
1922 tool (in the agent-based setting). Some of these fabricated code snippets may appear plausible but
1923 are not contextually appropriate or reflective of the real codebase, leading to failures in reasoning
1924 and incorrect results. Listing C.1 provides an example of this thinking process, where the LLM
1925 fabricates a code snippet that does not actually exist and relies on it to reason, ultimately producing
1926 a false result.
1927
1928 Listing 13: Example of failure type 1
1929
1930 The LLVMFuzzerTestOneInput function would pass the input data to libpng's parsing functions.
1931 ↪ The entry point for parsing a PNG file is typically png_read_png or similar. The fuzzer
1932 ↪ harness might look like:
1933
1934 int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
1935     png_structp png = png_create_read_struct(...);
1936     png_infop info = png_create_info_struct(png);
1937     // Set up error handling
1938     // Read the PNG data
1939     png_process_data(png, info, data, size);
1940     // ...
1941     png_destroy_read_struct(...);
1942     return 0;
1943 }
1944
1945 When processing the PNG data, the sCAL chunks are handled by png_handle_sCAL. The first
1946 ↪ sCAL chunk is processed normally, setting the info_ptr's valid flag. The second sCAL
1947 ↪ chunk triggers the else if (info_ptr != NULL ...) condition, leading to the target
1948 ↪ line.

```

1944
1945

C.2 FAILURE TYPE 2: INABILITY TO SYNTHESIZE VALID INPUT

1946
1947
1948
1949
1950
1951

By manually analyzing the responses from LLMs, we found that although LLMs possess detailed knowledge of valid target file format specifications (as verified through manual prompting), they often struggle to synthesize the correct file format and miss key specifications. Listing C.2 provides an example of LLM-generated code in which it synthesizes an invalid *IHDR* chunk due to neglecting important format specifications. This oversight leads to the generation of incorrect code that fails to meet the required format standards.

1952

1953
1954

Listing 14: Example of failure type 2

```
1955 import zlib
1956
1957 def generate_buf():
1958     # PNG signature
1959     png_sig = bytes.fromhex('89504E470D0A1A0A')
1960
1961     # Helper to create chunks
1962     def make_chunk(chunk_type, data):
1963         length = len(data).to_bytes(4, 'big')
1964         crc = zlib.crc32(chunk_type + data).to_bytes(4, 'big')
1965         return length + chunk_type + data + crc
1966
1967     # IHDR chunk (1x1 RGB)
1968     ihdr_data = bytes.fromhex('00000001 00000001 08020000 0000')
1969     ihdr = make_chunk(b'IHDR', ihdr_data)
1970
1971     ...
1972
```

1969
1970

C.3 FAILURE TYPE 3: FAIL TO REASON PLAUSIBLY

1971
1972
1973
1974
1975
1976
1977
1978
1979
1980

In some cases, we observed that LLMs struggle to make plausible connections between low-level conditions in the code and the higher-level constraints of the target file format. Instead of synthesizing the information and drawing valid inferences, they simply repeat the low-level conditions they encounter, failing to apply reasoning at a higher level. This issue is particularly common in non-reasoning LLMs. For example, in Listing C.3, the LLM simply reiterates the low-level conditions seen in the code, without translating these conditions into meaningful constraints related to the file format. This kind of response highlights the model’s inability to reason beyond the surface-level details.

1981

1982
1983

Listing 15: Example of failure type 3

```
1984 Based on the code analysis, we can now construct the full call chain and understand the
1985 → conditions needed to reach our target line in `png_handle_sCAL`:
1986
1987 Call chain:
1988 1. `LLVMFuzzerTestOneInput`
1989 2. `png_read_info` (via `png_read_png`)
1990 3. `png_push_read_chunk`
1991 4. `png_handle_sCAL`
1992
1993 To reach our target line in `png_handle_sCAL`, we need:
1994 1. A valid PNG header (first 8 bytes)
1995 2. The chunk type must be sCAL (`png_sCAL`)
1996 3. `info_ptr` must not be NULL
1997 4. `(info_ptr->valid & PNG_INFO_sCAL) != 0`
1998 5. `(png_ptr->mode & PNG_HAVE_IHDR) != 0`
1999 6. `(png_ptr->mode & PNG_HAVE_IDAT) == 0`
2000
2001 Here's the input generation code that satisfies these conditions:
2002 ...
2003
```

1998
1999

C.4 FAILURE TYPE 4: MISSING CONSTRAINTS IN COMPLEX CONTEXT

2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011

By manually analyzing the responses from LLMs, we found that for complex programs, LLMs sometimes focus narrowly on the target location and surrounding code, overlooking important constraints distributed across the broader context. Interestingly, this is similar to mistakes that humans sometimes make when dealing with complex code, focusing too much on a small part of the program and missing crucial elements that are spread across other sections. For example, Listing C.4 shows a reasoning process where an LLM concludes that the entry function parses an XML file directly from the input buffer. However, upon examining the start of the entry function, shown in Listing C.4, we see that it actually first processes an integer length field followed by the XML file. This oversight demonstrates the LLM’s failure to correctly connect and reason about the larger flow of the program, resulting in an incomplete understanding of the code’s behavior, which highlights the challenges LLMs face when handling intricate, context-dependent code and emphasizes the importance of maintaining a holistic view of the program during reasoning.

2012
2013
2014
2015
2016
2017
2018
2019
2020
2021

Listing 16: Example of failure type 4

```
In the LLVMFuzzerTestOneInput function, the fuzzer reads the input data, initializes the
→ parser, and processes the XML document. The docBuffer comes from xmlFuzzMainEntity,
→ which is part of the fuzzer's input. So, the input needs to be a valid XML document with
→ an element that has a prefix but no namespace URI.
```

2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051

Listing 17: Entry function in failure type 4 example

```
int
LLVMFuzzerTestOneInput(const char *data, size_t size) {
    static const size_t maxChunkSize = 128;
    xmlDocPtr doc;
    xmlParserCtxtPtr ctxt;
    xmlTextReaderPtr reader;
    xmlChar *out;
    const char *docBuffer, *docUrl;
    size_t docSize, consumed, chunkSize;
    int opts, outSize;

    xmlFuzzDataInit(data, size);
    opts = xmlFuzzReadInt();
    opts &= ~XML_PARSE_XINCLUDE;

    xmlFuzzReadEntities();
    docBuffer = xmlFuzzMainEntity(&docSize);
    docUrl = xmlFuzzMainUrl();
    if (docBuffer == NULL)
        goto exit;

    /* Pull parser */

    doc = xmlReadMemory(docBuffer, docSize, docUrl, NULL, opts);
    ...
```

D DETAILED ANALYSIS

D.1 NUMBER OF QUERIES BY LLMs IN AGENT-BASED SETTING

We analyze the number of queries emitted by LLMs in the agent-based setting and compare the query counts between reasoning and non-reasoning LLMs. The results, shown in Figure 7, reveal that reasoning LLMs emit significantly fewer queries than their non-reasoning counterparts. We speculate that this may be due to reasoning LLMs being more prone to hallucination.

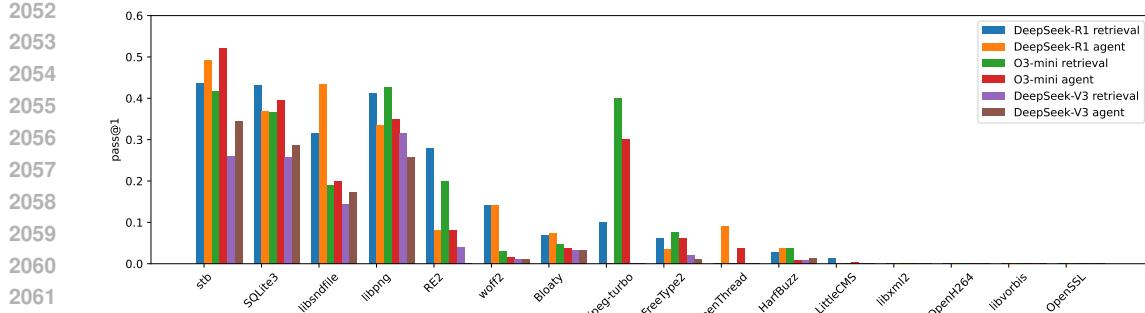


Figure 6: Pass@1 scores across different code repositories.

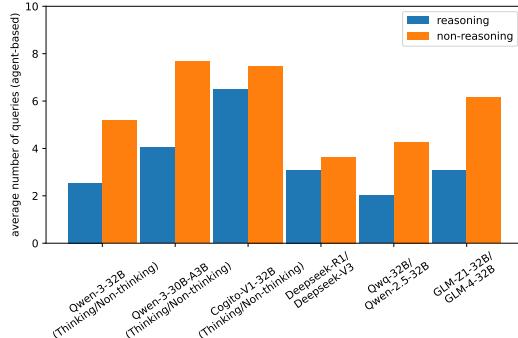


Figure 7: Number of queries emitted by LLMs.

D.2 PROBLEMS SOVLED BY DIFFERENT LLMs

In this section, we analyze the problems solved by different LLMs across all 500 problems in 5 trials, under both retrieval-based and agent-based settings. Exemplary Venn diagrams of the well-performing models are shown in Figure 8. The upset diagram is displayed in Figure 9, which further highlights the unique problem sets solved by each model. The results demonstrate that even though the overall performance of each model varies, they each manage to solve a distinct subset of problems that the others cannot. This observation underscores the complementary strengths of each model, as different models excel in addressing various aspects of the TTG problem. Such diversity in performance suggests that no single model is universally superior, and each has its own strengths when tackling different parts of the problem space.

E LIMITATIONS

The limitations and future directions of this paper include: (1) expanding the evaluation to multiple programming languages to assess LLMs’ comprehension across different programming languages, and (2) decomposing the TTG problem into more granular tasks that reflect specific aspects of the comprehensive abilities of LLMs. This will allow for a more detailed understanding of LLMs’ strengths and weaknesses in different contexts and domains, and help drive further improvements in their performance across a broader range of software engineering tasks.

F LLM USAGE

During the preparation of this manuscript, we made selective use of Large Language Models (LLMs), specifically, as a writing assistant for grammar correction and stylistic refinement. All

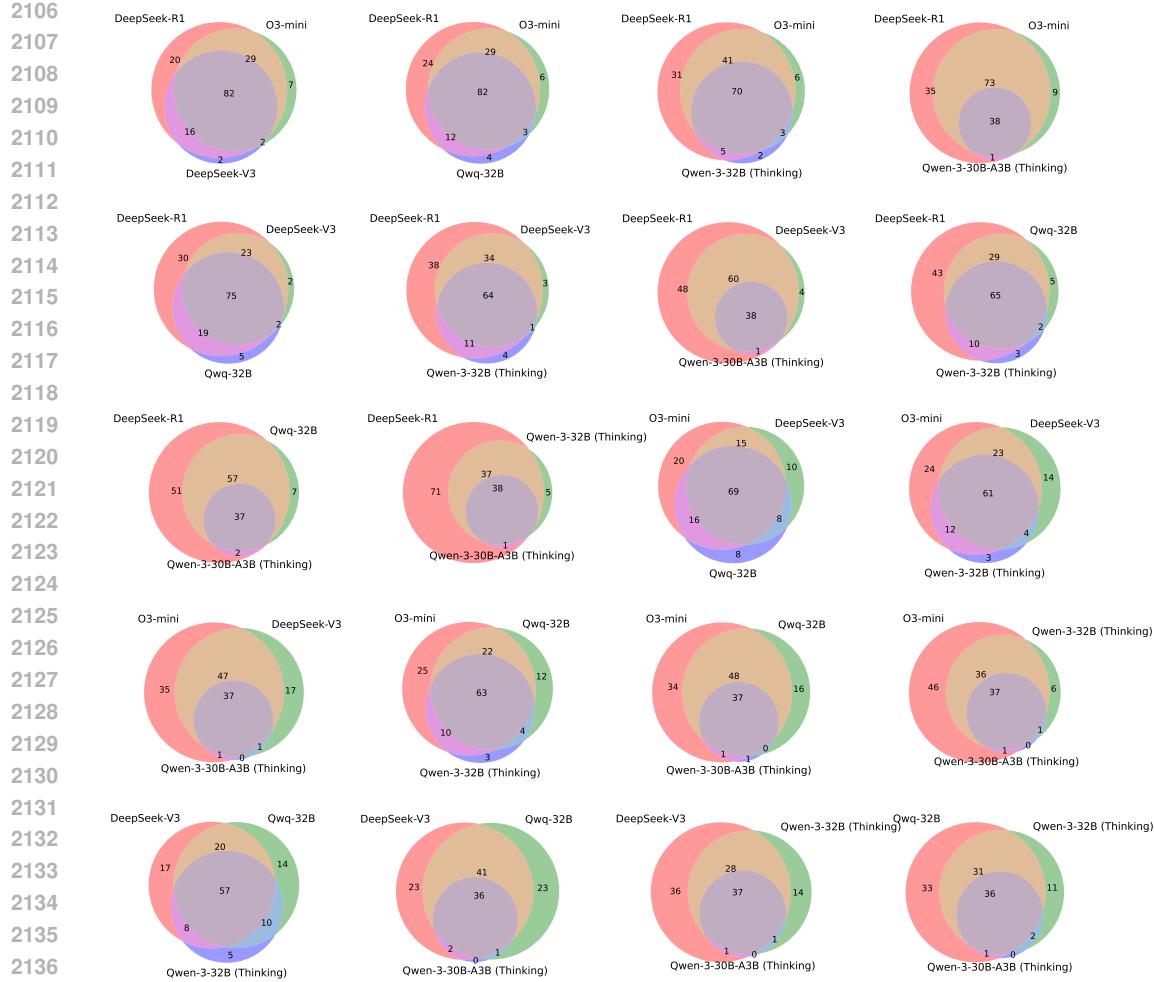


Figure 8: Venn diagrams of solved problems by different LLMs.

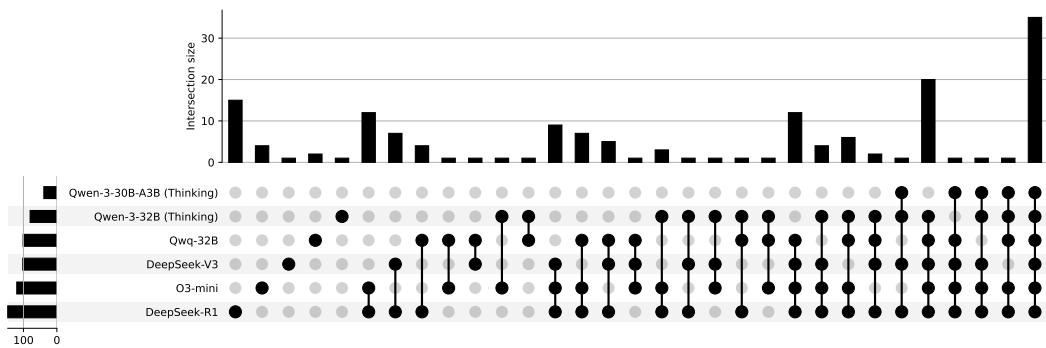


Figure 9: Upset diagram of solved problems by different LLMs.

2156
2157
2158
2159
scientific contributions were conceived and executed entirely by the authors. The LLM did not contribute to any substantive intellectual content.