How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation

Cen Zhang
Nanyang Technological University
Singapore

Yaowen Zheng*
Nanyang Technological University
Singapore

Mingqiang Bai
IIE, CAS; Sch of Cyber Security, UCAS
Beijing, China

Yeting Li
IIE, CAS; Sch of Cyber Security, UCAS
Beijing, China

Wei Ma
Nanyang Technological University
Singapore

Xiaofei Xie
Singapore Management University
Singapore

Yuekang Li
The University of New South Wales
Sydney, Australia

Limin Sun
IIE, CAS; Sch of Cyber Security, UCAS
Beijing, China

Yang Liu
Nanyang Technological University
Singapore

Abstract

Fuzz drivers are essential for library API fuzzing. However, automatically generating fuzz drivers is a complex task, as it demands the creation of high-quality, correct, and robust API usage code. An LLM-based (Large Language Model) approach for generating fuzz drivers is a promising area of research. Unlike traditional program analysis-based generators, this text-based approach is more generalized and capable of harnessing a variety of API usage information, resulting in code that is friendly for human readers. However, there is still a lack of understanding regarding the fundamental issues on this direction, such as its effectiveness and potential challenges.

To bridge this gap, we conducted the first in-depth study targeting the important issues of using LLMs to generate effective fuzz drivers. Our study features a curated dataset with 86 fuzz driver generation questions from 30 widely-used C projects. Six prompting strategies are designed and tested across five state-of-the-art LLMs with five different temperature settings. In total, our study evaluated 736,430 generated fuzz drivers, with 0.85 billion token costs ($8,000+ charged tokens). Additionally, we compared the LLM-generated drivers against those utilized in industry, conducting extensive fuzzing experiments (3.75 CPU-year). Our study uncovered that: 1) While LLM-based fuzz driver generation is a promising direction, it still encounters several obstacles towards practical applications; 2) LLMs face difficulties in generating effective fuzz drivers for APIs with intricate specifics. Three featured design choices of prompt strategies can be beneficial: issuing repeat queries, querying with examples, and employing an iterative querying process; 3) While LLM-generated drivers can yield fuzzing outcomes that are on par with those used in the industry, there are substantial opportunities for enhancement, such as extending contained API usage, or integrating semantic oracles to facilitate logical bug detection. Our insights have been implemented to improve the OSS-Fuzz-Gen project, facilitating practical fuzz driver generation in industry.

*Yaowen Zheng is the corresponding author.
IIE → Institute of Information Engineering, CAS → Chinese Academy of Sciences.
Email To: cen001@e.ntu.edu.sg & yaowen.zheng@ntu.edu.sg.

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Fuzz Driver Generation, Fuzz Testing, Large Language Model

ACM Reference Format:
Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In . ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Fuzz testing, aka fuzzing, has become the standard approach for discovering zero-day vulnerabilities. Fuzz drivers are necessary components for fuzzing library APIs since fuzzing requires a directly executable target program. Essentially, a fuzz driver is a piece of code responsible for accepting mutated input from fuzzers and executing the APIs accordingly. An effective driver must contain a correct and robust API usage since incorrect or unsound usage can result in extensive false positive or negative fuzzing results, incurring extra manual validation efforts or testing resources waste. Due to the high standard required, fuzz drivers are typically written by human experts, which is a labor-intensive and time-consuming process. For instance, OSS-Fuzz [19], the largest fuzzing framework for open-source projects, maintains thousands of fuzz drivers written by hundreds of contributors over the past seven years.

Generative LLMs (Large Language Models) have gained significant attention for their ability in code generation tasks [31, 33, 35, 40]. They are language models trained on vast quantities of text and code, providing a conversational workflow where natural language based queries are posed and answered. LLM-based fuzz driver generation is an attractive direction. On one hand, LLMs inherently support fuzz driver generation as API usage inference is a basic scenario in LLM-based code generation. On the other hand, LLMs are lightweight and general code generation platforms. Existing works [2, 22, 24, 26, 62–64], which generate drivers by learning API usage from examples, requires program analysis on examples, while LLM-based generation can mostly work on texts. This offers enhanced generality which facilitates not only the application on massive quantity of real-world projects but also the utilization of

learning inputs in different forms. Various sources of API usage knowledge such as documentation, error information, and code snippets can be seamlessly integrated in text form, benefiting the generation. Moreover, LLMs can generate human-friendly code. While some research efforts have been devoted to LLM-based code generation tasks [15, 23, 25, 29, 32, 38, 42, 49], none of them can provide a fundamental understanding on this direction.

To address this gap, we conducted an empirical study for understanding the effectiveness of zero-shot fuzz driver generation using LLMs. Note that our primary goal is to understand the basics towards generating "*more*" effective fuzz drivers, rather than generating "*more effective*" fuzz drivers. This is because creating effective drivers for more targets is a more fundamental issue than improving existing ones. Overall, four research questions are studied:

- **RQ1** To what extent can current LLMs generate effective fuzz drivers for software testing?
- **RQ2** What are the primary challenges associated with generating effective fuzz drivers using LLMs?
- **RQ3** What are the effectiveness and characteristics for different prompting strategies?
- **RQ4** How do LLM-generated drivers perform comparing to those practically used in the industry?

To answer these RQs, we assembled a dataset of 86 fuzz driver generation questions collected from 30 widely-used C projects from OSS-Fuzz projects. Each question represents a library API for which a corresponding fuzz driver is needed to conduct effective fuzz testing. We devised six prompt strategies, taking into account three key factors: the content of the prompts, the nature of interactions between the strategies and models, and the repetition of the entire query process. Our evaluation encompassed five state-of-the-art LLMs with five different temperature settings. The assessed LLMs included closed-source LLMs such as gpt-4-0613 [37], gpt-3.5-turbo-0613 [36], and text-bison-001 [20], as well as open-source LLMs optimized for code generation, namely, codellama-34b-instruct [4] and wizardcoder-15b-v1.0 [58]. For a rigorous assessment, we developed an evaluation framework automatically validating the generated drivers based on the results of compilation and short-term fuzzing, and manually crafted checkers on API usage semantic correctness. In total, 736,430 fuzz drivers, at the cost of 0.85 billion tokens ($8,000+ charged tokens, 0.17/0.21 billion for gpt-4-0613/gpt-3.5-turbo-0613), were evaluated. Besides, comparison with manually written drivers in industry on code and fuzzing metrics, *e.g.*, 24-hour fuzzing experiments (3.75 CPU-year), are conducted to seek practical insights.

The overall implications for the effectiveness of using LLM to generate fuzz drivers are two-fold. On one hand, LLMs have demonstrated outstanding performance in evaluated configurations[1], suggesting a strong potential for this approach. For instance, the optimal configuration can address 91% questions (78/86) and all top 20 configurations can address at least half of the questions. On the other hand, resolving a question means successful generation of at least one effective fuzz driver for the assessed API, which does not necessarily imply full practicality. For high automation and usability, three challenges have been identified: ❶ Improving the success rate to reduce generation costs. Although most questions can be resolved by LLMs, the cost can be exceptionally high. Typically, 71% of questions are resolved by repeating the entire query process at least five times and 45% require repeating the process ten times. By enhancing their accuracy, significant financial costs for automation can be saved. ❷ Ensuring semantic correctness in API usage. Occasionally, validating the effectiveness of a generated driver requires the understanding of API usage semantics. Failed to do so can result in ineffective fuzzing with false positive or negative results. In our evaluation, this requirement was observed in 34% of the assessed APIs (29/86), impeding practical application. ❸ Addressing complex API dependencies. 6% of questions (5/86) cannot be resolved by any evaluated configurations since their drivers' generation requires nontrivial preparation of the API execution contexts, which cannot be appropriately hinted by any collected usage information. For example, some drivers require a standby network server or client to be created for interacting with the target API. These are typical cases representing complex real-world testing requirements which deserves exploration of advanced solutions.

Prompt strategy, temperature, and model are key factors considerably affect the overall performance. Our evaluation suggests that the dominant strategy is the one incorporating three key designs: repeatedly query, query with extended information, and iterative query. Comparing with naive strategy, its question resolve rate soars from 10% to 91%. Evaluation with lower temperature settings, especially below the threshold of 1.0, have higher performance. This is intuitive since lower temperatures lead to more consistent and predictable outputs, which fits the goal of generating an effective fuzz driver. Besides, the optimal temperature setting in our evaluation is 0.5. As for models, gpt-4-0613, wizardcoder-15b-v1.0 are the best closed-source, open-source models, respectively.

Fundamentally, LLMs struggle to generate fuzz drivers which require complex API usage specifics. We identified three beneficial designs that have distinct characteristics: ❶ repeatedly queries. When the configurations are stronger, the benefits of repetition become higher. Besides, the benefits significantly drop after the first few rounds. A suggested repetition value is 6. ❷ query with extended information. Adding API documentation is less helpful while adding example snippets can help significantly. Specifically, test/example files of the target project or its variants are high-quality example sources; ❸ iterative queries. It adds a cyclic driver fix progress after the initial query, which improves LLM's performance through its step-by-step problem-solving approach and a more thorough utilization of existing usage. Besides, all the above designs will significantly increase the token cost. In comparison to OSS-Fuzz drivers, LLM-generated drivers demonstrated comparable fuzzing outcomes. However, since LLMs tend to generate fuzz drivers with minimal API usages, significant room is still left for improving generated drivers, such as expanding API usage and incorporating semantic oracles.

To further translate our research insights into practical values, part of our prompting strategies, including checking, categorizing, and fixing driver with runtime errors, have been implemented into OSS-Fuzz-Gen [49], the largest LLM-based fuzz driver generation framework operated by Google OSS-Fuzz team, facilitating the continuous fuzzing of real-world projects.

In summary, our contributions are:

[1]A configuration stands for a combination of <Model, Prompt Strategy, Temperature>.

```
1 int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
2         // Input_Arrangement(Data, Size, ...);
3         Init_Context_And_Arguments(Data, Size, ...);
4         Call_Target_API(...);
5         // Extended_API_Usage(...);
6         // Semantic_Oracle_Checks(...);
7         Resource_Cleaning(...);
8         return 0;
9 }
```

Figure 1: Key Components of A Fuzz Driver.

- we conducted the first in-depth study on the effectiveness of LLM-based fuzz driver generation, which showcases the potentials and challenges of this direction;
- we designed and implemented six driver generation strategies. They are evaluated in large scale, with a systematic analysis on the effectiveness, the pros and the cons;
- we compared generated drivers with industrial used ones, and summarized the implications on future improvements.
- we ported our strategies to improve the largest industrial fuzz driver generation framework, facilitating the continuous fuzzing of hundreds open-source projects.

2 Preliminaries

Fuzz Driver Basics. The key components of a fuzz driver are illustrated in Figure 1. A typical fuzz driver has three necessary parts: prerequisites initialization (line 3), execution (line 4), and post-cleaning (line 7). Besides, there are three optional parts commetned in lines 2, 5, and 6 that can improve a driver's effectiveness. Line 2 part improves a driver by proper input arrangement such as rejecting too short or too long inputs, interpreting input data as multiple testing arguments, etc. Line 5 part enables a driver to call more APIs which triggers more program behaviors during fuzzing. Finally, line 6 part adds semantic oracles for detecting logical bugs. These oracles are similar to `assert` statements in unit tests, aborting execution when certain program properties are unsatisfied. Since a driver will be repeatedly executed with randomly mutated input, there is a high requirement on its correctness and robustness. Incorrect or unrobust usage can lead to both false positives and negatives. For instance, if a driver failed to feed the mutated data into the API, its fuzzing can never find any bug. Or if an API argument is incorrectly initialized, false crashes may be raised.

Minimal Requirements of Effective Fuzz Drivers. The minimal requirements covers the line 3,4, and 7 of Figure 1, which mainly include correctly initializing the arguments and satisfying necessary control flow dependencies. Argument initialization can be one of the following cases (in the order of simplicity): ❶ **C1**: If the argument value can be any value or should be naive values like `0` or `NULL`, a variable is declared or a literal constant is used directly; ❷ **C2**: If the argument is supposed to be a macro or a global variable that is already defined in common libraries or the target API's project, it is located and used; ❸ **C3**: If creating the argument requires the use of common library APIs, such as creating a file and writing specific content, common practices are followed; ❹ **C4**: If initializing the argument requires the output of other APIs within the project, those APIs are initialized first following the above initialization cases.

3 Methodology

3.1 Design of Prompt Strategies

Figure 2 illustrates our designed prompt strategies. From left to right, the figure first provides a tabular overview for all proposed strategies, then details two types of prompt templates involved, and lastly maps the templates to concrete query examples. Note that the listed examples are simplified for demonstration purposes, while unmodified real-world examples for each strategy can be found at [1].

Key Designs. As shown in the top-left side of Figure 2, there are three key designs for prompt strategies, including query with different types of API information, query repeatedly, and query iteratively. Design I aims for understanding the generation effectiveness given different API information as query contexts. The information is divided as two types: the basic API information and the extended. The former includes fundamental information such as header file name and API declaration, which are precisely specified and generally accessible in library API fuzzing scenario, while the latter requires additional resources like API-specific documentation or usage example code snippets, whose quality and availability vary for different targets. To account for the inherent randomness in LLM output generation, design II, repeatedly query, is introduced. Given repetition time as value K, the entire query process of a strategy will be repeated K times ($K \geq 1$), generating K independent drivers. The maximum value of K is set as 40 in our study. This is an empirically value we believe is comprehensive enough to understand the effectiveness of repetition. Design III is used to understand the effectiveness of different query workflows. The driver generation in non-iterative strategies follows a one-and-done manner where the final driver is synthesized via a single query without further refinement. Iterative strategies have a generate-and-fix workflow. If the driver generated in first query fails to pass the automatic validation, subsequent fix prompts are composed based on the error feedback and queried. The fix continues until the driver passes validation or a pre-defined maximum number of iterations is reached. The iteration is limit as five in our evaluation.

Acronym. Strategies are named by concatenating the abbreviations of the three key designs. For all strategies, there is a suffix "K" indicating that the repetition times of their query process. If a strategy name contains "ITER", it is an iterative prompt strategy. Otherwise, it is non-iterative. Besides, different combinations of API information used in generation prompt have different abbreviations. As shown in the bottom-left side of the Figure 2, there are four different combinations: the **NAIVE** query context (abbr as NAIVE, ①), the **BA**sic query **C**on**T**e**X**t (abbr as BACTX, ① + ②), extending API **DOC**umentation to basic **C**on**T**e**X**t (abbr as DOCTX, ① + ② + ④), and extending example **U**sa**G**e code snippets to the basic **C**on**T**e**X**t (abbr as UGCTX, ① + ② + ③). Lastly, the prefix ALL in ALL-ITER-K indicates that its prompts can be the prompt designed in any other strategies.

NAIVE-K & BACTX-K. Both two strategies only use basic API information in query and non-iterative workflow. Their only difference is the richness of the prompt context information. Specifically, NAIVE-K directly asks LLMs to implement the fuzz driver solely based on a specified function name, while BACTX-K provides a basic description of the API. In prompts of BACTX-K, it first indicates the task scope using `#include` statement, then provides the function

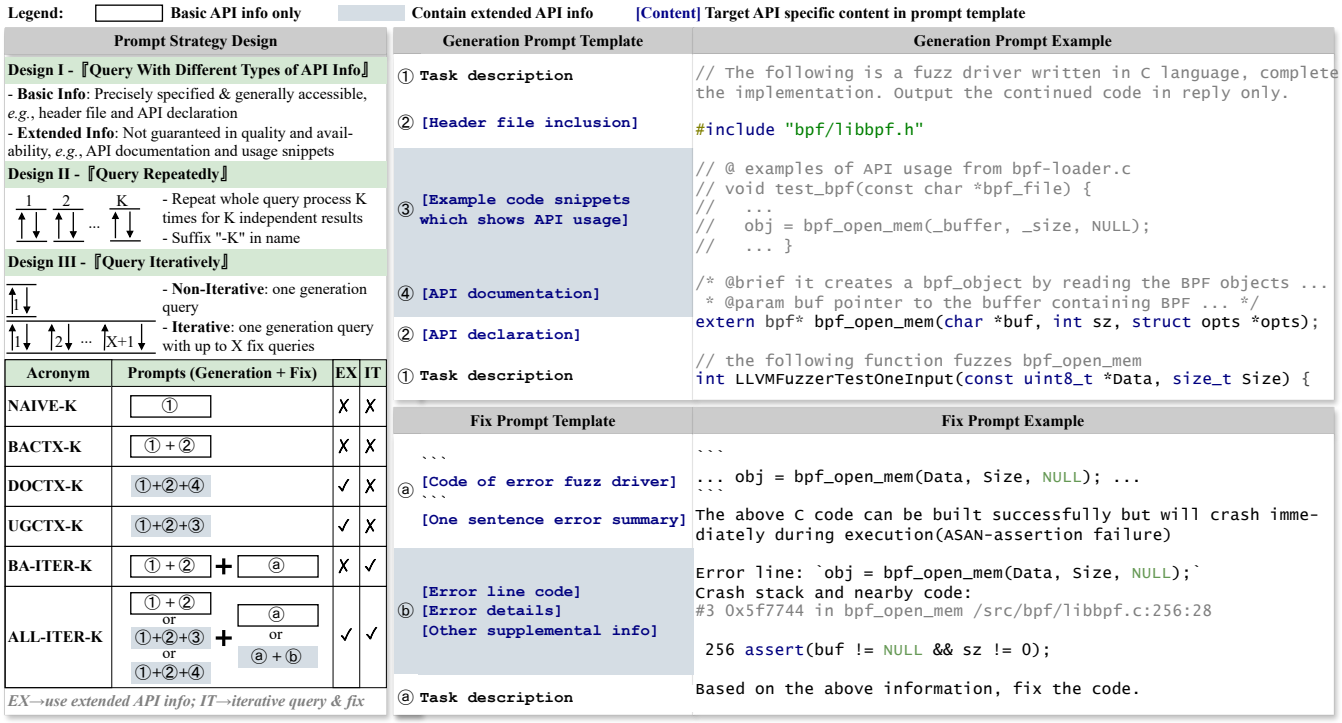


Figure 2: Prompt Strategies Overview. **K is 1 or 40 in our evaluation and X is 5. The examples are simplified for demonstration purpose. In the fix prompt example, the driver error is caused by missing check of Size > 0 before calling the API, and the nearby code of #3 stack frame hints the error.**

declaration, and finally requests implementation. The declaration is extracted from the Abstract Syntax Tree (AST) of the header file, including both the signature and argument variable names.

DOCTX-K & UGCTX-K. These two strategies are extended from BACTX-K by adding extended usage information in query. Their effectiveness represents the effects of two types of extended information: API documentation and example code snippets. Note that, for DOCTX-K, not all APIs have associated documentation (49/86 questions in our study). The documentation of 20 questions was automatically extracted from the header files, while the remaining 29 were manually collected from sources like project websites, repositories, and developer manuals. For UGCTX-K, example code snippets of an API are collected as follows: ❶ retrieving the files containing usage code via `SourceGraph cli` [44]. This is a keyword search among all public code repositories including Github, Gitlab, etc. The crawling command is `src search -json "file:.*c lang:c count:all API"` where `API` should be replaced by the target API name. ❷ identifying and excluding fuzz drivers by removing the files containing function `LLVMFuzzerTestOneInput`. ❸ extracting all functions directly calling the target API as example code snippets via `ANTLR` based source code analysis. ❹ deduplicating the snippets by if the Jaccard Similarity [21] of any two snippets ≥95%. UGCTX-K will randomly use one snippet in the prompt. For snippet that was too long to be included into prompt, it is truncated line by line until satisfying the token length limitation.

BA-ITER-K & ALL-ITER-K. Iterative strategies have two types of prompt templates for initial generation query and subsequent fix query. The initial generation prompt can be either of BACTX-K's, DOCTX-K's, and UGCTX-K's. As for fix queries, we have designed seven fix templates to address seven prevalent types of errors in the generated drivers. They follow one general fix template shown in Figure 2 but are filled with error type specific details. Due to the page limit, we discuss the key concepts of these fix prompts, leaving the detailed designs and examples in [1]. These errors are of compilation errors (1/7), linkage errors (1/7), and fuzzing runtime errors (5/7). The error information (abbr of `[One sentence error summary]`, `[Error line code]`, and `[Error details]`) for the first two error types can be programmatically retrieved from the compiler. According to different abnormal behaviors observed in fuzzing, fuzzing runtime errors have five subtypes, including memory leakage, out-of-memory, timeout, crash, and non-effective fuzzing (no coverage increase in one-minute short-term fuzzing). The error information of runtime errors are retrieved by extracting the crash stacks and sanitizer summary from `libfuzzer` logs. Lastly, for the errors that can locate its error line, we further infer its root cause API and fill API information like declaration, documentation, or usage snippets into `[Other supplemental information]` in fix prompt. For simplicity, root cause API is identified by naively finding the last executed API based on the error line located.

Note that iterative strategies exclusively utilize automated checkers, ensuring that the manually crafted semantic checkers described in Section 3.2 are only used for thorough evaluation of the effectiveness of strategies. The principal distinction between the two iterative strategies, BA-ITER-K and ALL-ITER-K, lies in the scope of information utilized within the queries. BA-ITER-K confines its use to only basic API details and error information, whereas ALL-ITER-K encompasses all available information. As shown in

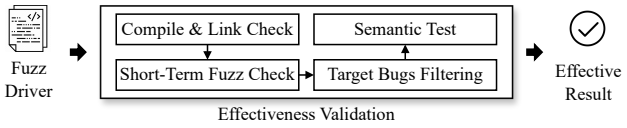


Figure 3: Fuzz Driver Effectiveness Validation Process.

Figure 2, this leads to multiple options for the included extended information. The *ALL-ITER-K* strategy selects options randomly.

3.2 Evaluation Framework

Evaluation Question Collection. One question used in evaluation is simply designed as generating fuzz drivers for one given API. However, not all APIs are suitable to be set as questions. Naively collecting all APIs from projects will lead to the creation of meaningless or confusing questions which influences the evaluation result. Specifically, some APIs, such as `void libxxx_init(void)`, are meaningless fuzz targets since the code executed behind the APIs can not be affected by any input data. Some APIs can only be supplemental APIs rather than the main fuzzing target due to the nature of their functionalities. For example, given two APIs `object *parse_from_str(char *input)` and `void free_object(object *obj)`, feeding the mutated data into `input` is apparently a better choice than feeding a meaningless pointer to the `obj` argument. However, calling the latter API when fuzzing the former is meaningful since ❶ it may uncover the hidden bug when the former does not correctly initialize the `object *` by freeing the members of `object`; ❷ it releases the resources allocated in this iteration of fuzzing, which prevents the falsely reported memory leak issue.

To guarantee the collected APIs are qualified and representative, we collected the core APIs of existing fuzz drivers from OSS-Fuzz projects. A driver's core APIs are identified using the following criteria: ❶ they are the target APIs explicitly pointed out by the author in its driver file name or the code comments, *e.g.*, `dns_name_fromwire` is the core API of driver *dns_name_fromwire.c*; ❷ otherwise, we pick the basic APIs as the core rather than the supplemental ones. For example, we picked the former between parse and use/free APIs. For the fuzz drivers which are composite drivers fuzzing multiple APIs simultaneously, we identified multiple core APIs from them. Specifically, we randomly selected 30 projects from OSS-Fuzz (commit `135b000926`) C projects, manually extracted 86 core APIs from 51 fuzz drivers. Full list of questions are post at [1].

Effectiveness Validation Criteria. Assessing the effectiveness of a generated fuzz driver is complex since identifying both false positives (bugs caused by the driver code) and negatives (can never find bugs given its incorrect usage) rely on the understanding of API usage semantics. Figure 3 is a streamlined four-step semi-automatic validation process: ❶ Use a compiler to check for grammatical errors in the driver code. ❷ Observe the driver in a one-minute fuzzing session starting with no initial seed. It is ineffective if it either fails to show coverage progress or reports any bugs. The assumption behind is that, given a poor fuzzing setup, neither the zero coverage progress nor the quick identification of bugs for a well-tested API are normal behaviours. Considering that this criteria can still lead to incorrect validation, two additional steps are introduced for result refinements. ❸ If the driver reports bugs in ❷, we filter the true bugs contained inside. This is done by first collecting target true bugs via ten cpu-day fuzzing using existing OSS-Fuzz drivers, then manually building filters based on the root causes of these bugs. ❹ For drivers reporting no bugs after ❸, we check whether they are substantially testing the target API or not. To this end, we write API-specific semantic tests to detect common ineffective patterns observed in LLM-generated fuzz drivers. The tests include verifying the target API is called for all 86 questions, checking the correct usage of fuzzing data to key arguments for 8 questions (such as fuzzing file contents instead of file names), ensuring critical dependent APIs are invoked in 16 questions, and confirming necessary execution contexts are prepared for 5 questions (for instance, having a standby server process available for testing client APIs). We implement these tests by injecting hooking code into the driver. For more details on these tests, please refer to our website [1].

Evaluation Configuration. In this paper, a configuration represents a specific combination of the three factors: the LLM used, the prompt strategy employed, and the selected temperature setting, abbr as <model, prompt strategy, temperature>. As shown in Table 1, we evaluated six prompt strategies on five LLMs with five different temperatures. A configuration's top_p is set as its model's default value. And the system role [34] is set as "`You are a security auditor who writes fuzz drivers for library APIs.`".

4 Overall Effectiveness (RQ1)

Table 1 presents the results of all evaluated configurations. The principal data displayed in the table are the question solve rates, formatted as X/Y, where X denotes the number of questions a language model successfully solves, and Y represents the total number of questions presented. A configuration is considered to have solved a question if at least one effective fuzz driver has been generated. For gpt-4-0613, temperature 2.0 results were incomplete due to the services' slow response time in extreme temperature settings [1]. For the text-bison-001, Google's query API limits the requests with a temperature setting above 1.0. Nevertheless, given the poor or even zero performance of all other models with a temperatures setting 2.0, the data absence does not substantially affect our evaluation. This table only lists the number of solved questions while [1] posts full evaluation details of each model such as success rate per question.

Overall, the results offer promising evidence of the practicality of utilizing language model-based fuzz driver generation. The optimal configurations, namely <gpt-4-0613, ALL-ITER-K, 0.5>, achieved impressive success rates, effectively generating fuzz drivers that solved about 91% (78/86) questions. Moreover, three out of five LLMs assessed – including an open-source option – and half of the strategies explored can resolve over half of the questions.

The substantial variation in success rates across different configurations underscores the significant influence of the three factors. By analyzing the data, we observe that results can greatly fluctuate when varying a single factor – such as changing the temperature in a row, or switching models or prompting strategies in a column. For example, <gpt-3.5-turbo-0613, NAIVE-1, 0.0> failed to solve any questions, whereas <gpt-3.5-turbo-0613, ALL-ITER-K, 0.0> managed to correctly address 76% (65/86) of them. This indicates that achieving a high solve rate relies heavily on avoiding suboptimal combinations of factors. Given that the table is sorted to reflect performance trends, the better outcomes tend

Table 1: Overall Evaluation Result. **K represents 40, "-" means failed to retrieve full query results or not applicable for the given model.**

		Temperature				
	Strategy, Model	0.0	0.5	1.0	1.5	2.0
NAIVE-1	gpt-4-0613	9/86	9/86	9/86	0/86	-.
	gpt-3.5-turbo-0613	0/86	1/86	0/86	0/86	0/86
	wizardcoder-15b-v1.0	3/86	1/86	1/86	0/86	0/86
	text-bison-001	2/86	2/86	1/86	-.	-.
	codellama-34b-instruct	0/86	0/86	0/86	0/86	0/86
NAIVE-K	gpt-4-0613	12/86	30/86	30/86	5/86	-.
	gpt-3.5-turbo-0613	0/86	6/86	8/86	8/86	0/86
	wizardcoder-15b-v1.0	3/86	8/86	11/86	1/86	0/86
	text-bison-001	2/86	5/86	5/86	-.	-.
	codellama-34b-instruct	0/86	1/86	3/86	0/86	0/86
BACTX-K	gpt-4-0613	29/86	41/86	41/86	21/86	-.
	gpt-3.5-turbo-0613	12/86	29/86	30/86	24/86	1/86
	wizardcoder-15b-v1.0	7/86	23/86	25/86	17/86	0/86
	text-bison-001	7/86	13/86	15/86	-.	-.
	codellama-34b-instruct	0/86	1/86	11/86	0/86	0/86
DOCTX-K	gpt-4-0613	29/86	40/86	41/86	22/86	-.
	gpt-3.5-turbo-0613	11/86	22/86	29/86	24/86	1/86
	wizardcoder-15b-v1.0	7/86	24/86	25/86	12/86	0/86
	text-bison-001	9/86	14/86	14/86	-.	-.
	codellama-34b-instruct	0/86	9/86	13/86	1/86	0/86
UGCTX-K	gpt-4-0613	55/86	63/86	62/86	26/86	-.
	gpt-3.5-turbo-0613	30/86	47/86	43/86	31/86	0/86
	wizardcoder-15b-v1.0	39/86	50/86	48/86	13/86	0/86
	text-bison-001	21/86	27/86	38/86	-.	-.
	codellama-34b-instruct	0/86	8/86	21/86	0/86	0/86
BA-ITER-K	gpt-4-0613	56/86	57/86	62/86	23/86	-.
	gpt-3.5-turbo-0613	32/86	47/86	43/86	28/86	2/86
	wizardcoder-15b-v1.0	8/86	24/86	37/86	13/86	0/86
	text-bison-001	9/86	15/86	20/86	-.	-.
	codellama-34b-instruct	6/86	28/86	22/86	0/86	0/86
ALL-ITER-K	gpt-4-0613	77/86	78/86	76/86	25/86	-.
	gpt-3.5-turbo-0613	65/86	68/86	65/86	37/86	0/86
	wizardcoder-15b-v1.0	41/86	48/86	53/86	11/86	0/86
	text-bison-001	21/86	37/86	42/86	-.	-.
	codellama-34b-instruct	13/86	18/86	26/86	1/86	0/86

to cluster in 'green areas', highlighting configurations where all contributing factors are well-adjusted.

4.1 Analysis of Effectiveness Factors

Prompt Strategies. The observed impacts of different prompting strategies exceeded our initial expectations during their design phase. A comparison between NAIVE-1 and ALL-ITER-K showcases a dramatic improvement in optimal question solve rates, soaring from 10% to 90%, emphasizing the critical role of prompt design on tool effectiveness. To better understand the performance trends, Table 1 presents the prompting strategies ranked by their overall effectiveness. The trends in the results are intuitive: **in general, strategies that more comprehensively leverage available information tend to yield superior results**. For example, the strategy UGCTX-K markedly outperforms BACTX-K. This can be attributed to UGCTX-K's inclusion of example code snippets that illustrate certain usage of the target API. A notable performance discrepancy is also seen when comparing BA-ITER-K with BACTX-K. Despite starting with the same initial information, BA-ITER-K significantly surpasses BACTX-K. The reason for this performance difference lies in BA-ITER-K's iterative method – collecting debugging information to guide the model to fix the previous fuzz driver if it is ineffective. Among all the strategies, ALL-ITER-K stands out as the most effective across different combinations of temperature settings and models. This makes sense considering that ALL-ITER-K not only incorporates all extended API information but also adopts a recursive problem-solving methodology. Conclusively, its design leads to the superior performance in our evaluation. The detailed analysis of these strategies are discussed in Section 7.

Temperatures. Table 1 clearly demonstrates that **configurations with a temperature setting of 0.5 tend to achieve the highest success rates**. In contrast, models under a temperature setting above 1.0 experience a noticeable drop in performance. Interestingly, it appears that **in general, lower temperatures, especially below the threshold of 1.0, show substantial performance advantage compared to models operating at higher temperatures**. A surprising outcome is that models with 0.0 temperature perform remarkably well. For instance, both <gpt-4-0613, ALL-ITER-K, 0.0> and <gpt-3.5-turbo-0613, BA-ITER-K, 0.0> stand out as second-best configuration when compared across the various temperature settings. These results are reasonable considering the nature of fuzz driver generation task. With a lower temperature setting, models tend to generate more consistent and predictable outputs, which benefits the synthesis of high-quality code. High temperatures, while fostering creativity and randomness, may not provide any notable advantages in this context. Specifically, these features are either substituted by the randomness contained in prompt strategies or deemed irrelevant by the assessment criteria. For example, a prompting strategy like ALL-ITER-K inherently contains a built-in search process that brings the randomness from model input. And the evaluation strictly assesses the quantity of effective drivers without considering the API usage diversity. This criteria fits our evaluation goal, but discounts the creative diversity that could be introduced by higher temperatures.

Open-Source LLMs vs Closed-Source LLMs. As commonly understood in the industry, closed-source LLMs tend to outperform their open-source counterparts. Among these, gpt-4-0613 is considered the front-runner in terms of generation capabilities. Following closely behind is gpt-3.5-turbo-0613, which offers a cost-effective alternative due to its significantly lower token pricing. However, it's worth noting that in the open-source domain, wizardcoder-15b-v1.0 has made remarkable strides, even surpassing Google's closed-source model, text-bison-001. While wizardcoder-15b-v1.0 is nearly on par with gpt-3.5-turbo-0613, certain performance gaps can still be observed, but it stands as a commendable achievement for an open-source model.

4.2 How Far Are We to Total Practicality?

The above evaluation indicates that with the optimal configuration <gpt-4-0613, ALL-ITER-K, 0.5>, the LLM can solve 91% predefined questions. In other words, it can produce at least one effective fuzz driver for 78 out of 86 APIs examined. However, this does not necessarily mean that LLMs are ready to be used in production.

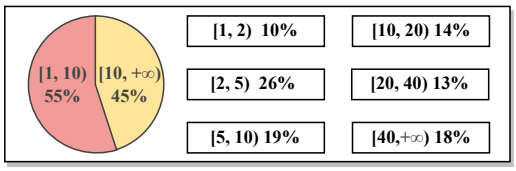


Figure 4: Question Cost Distribution For All Configurations on Resolved Questions. $Cost\ of\ A\ Question = \frac{\#\ of\ Queries}{\#\ of\ Solutions}$.

Upon further examination of our APIs, we identified three primary challenges and detailed them as follows.

C1: High Token Cost in Fuzz Driver Generation. Though many configurations have shown a high rate of successful problem resolution, our analysis indicates the results come with substantial costs. The data in Figure 4 details the percentage of high cost questions for all evaluated configurations. Remarkably, it reveals that on average, resolving 45% questions entail costs exceeding 10. This suggests that for 50% of the resolved questions, a prompt-based strategy may yield just one effective fuzz driver after repeating the entire query process 10 times or more. When considering only the questions with costs surpassing 20 or even 40, the percentages remain notable at 31% and 18%, respectively. These findings underscore a strong incentive for further research into cost reduction techniques. Reducing costs is not only a practical concern with direct financial consequences but also essential for improving the efficiency of LLM-based fuzz driver generation.

C2: Ensuring Semantic Correctness of API Usage. In our evaluation, we found that there is a discrepancy for approximately 34% (29/86) of the APIs – assuming LLMs can successfully create at least one effective fuzz driver for each in the evaluation setting, this success cannot be translated into the full automation of fuzz driver generation for them. The issue at hand lies in the potential misuse of APIs within the generated drivers, which requires validation to ensure semantic correctness. For example, LLMs may incorrectly initializing the argument of an API, such as passing a mutated filename to the API instead of passing a created file first and then mutating its content for fuzzing, or missing some condition checks before calling API. In our evaluation process, we manually implemented semantic checkers to identify such API misuses for accurate assessment (details on our semantic checkers are provided in Section 3.2). However, fully automating the validation of semantic correctness remains a significant hurdle. Consequently, even though it is feasible to generate effective fuzz drivers with the help of these LLMs, distinguishing them from the ineffective ones can be problematic due to the absence of automated methods for validating semantic correctness. This challenge underscores the need for developing robust techniques to automatically ensure the semantic accuracy of generated fuzz drivers before they can be reliably deployed in production.

C3: Satisfying Complex API Usage Dependencies. Overall, there are five questions cannot be resolved by any assessed configurations. These questions are challengeable since their driver generation requires the deep understanding of specific contexts. For instance, generating the driver for `tmux` [11] requires the construction of various concepts, such as session, window, pane, etc, and their relationships. Similarly, for network-related questions [5, 9], a standby network server or client is required to be created before calling the target API. The effective drivers can only be generated by respecting these specific contextual requirements.

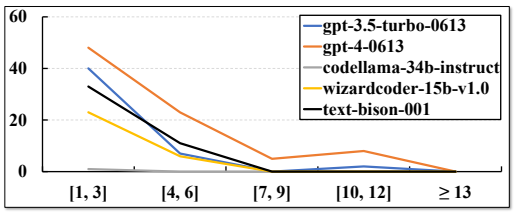


Figure 5: Average Query Success Rate Per Question Complexity Score Bucket. **Data: All BACTX-K (K = 40) configurations.**

While LLM-based generation has shown promising potential, it still faces certain challenges towards high practicality.

5 Fundamental Challenge (RQ2)

5.1 Links Between Question and Performance

To investigate the core difficulties in generating fuzz drivers with LLMs, we scrutinized the outcomes of the BACTX-K strategy. This strategy is a proper starting point for understanding the fundamental challenges since it merely uses generally accessible information and has simple query workflow. In Figure 5, **there is a clear inverse proportion relationship between the query success rate and the complexity of a question, irrespective of the used models and temperatures**. The complexity of a question is measured by first constructing the minimal fuzz driver of each question and then quantifying the API specific usage contained in the minimized code. A minimal effective driver for a question is created based on the OSS-Fuzz driver by removing the unnecessary part of the code and replacing the argument initialization into a simpler solution according to the cases enumerated in Section 2. Then the complexity is quantified as the sum of the count of the following elements inside code: ❶ unique project APIs; ❷ unique common API usage patterns; ❸ unique identifiers including non-zero literals and project global variables excluding the common API usage code; ❹ branches and loops excluding the common API usage code. Note that all branches of one condition will be counted as one. Overall, ❶, ❸ measure API specific vocabularies while ❹ for API specific control flow dependencies'. We put detailed calculation examples at [1].

Considering the generation process, it is intuitive that **LLMs' performance degrades when the complexity of target API specific usage increases**. To generate effective drivers, LLMs should at least generate code satisfying minimal requirements. In other words, they must accurately predict the API argument usage and control flow dependencies. However, this is challenging since LLMs cannot validate their predictions against documentation or implementations as humans do. It is reasonable to assume that LLMs have learned the language basics and common programming practices due to their training on vast amounts of code. But the API specific usage, such as the semantic constraints on the argument, cannot be assumed. On one hand, there may only have limited data about this in training. On the other hand, details can be lost during preprocessing or the learning stage while the accurate generation is required. Therefore, the more API usage a LLM needs to predict, the

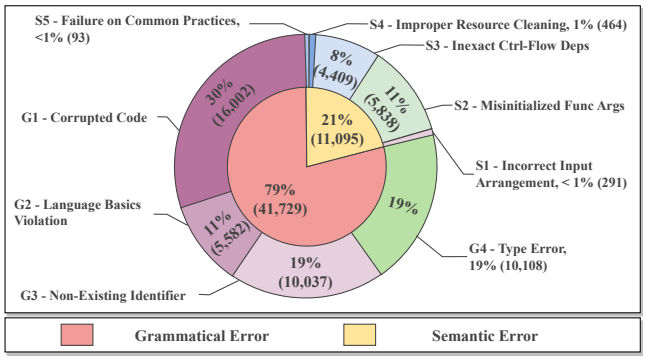


Figure 6: Failure Taxonomy. **Data: BACTX-K (K = 40) configurations.**

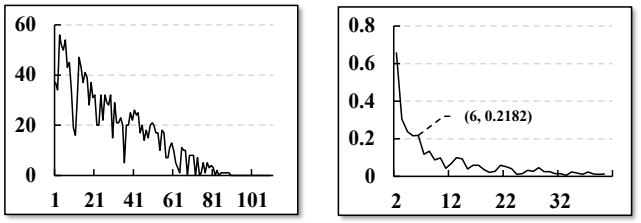
greater the likelihood of errors, particularly for less common usages that do not follow the mainstream design patterns or have special semantic constraints. Such situations are common in C projects, whose APIs often contain low-level project-specific details.

The performance of LLM-based generation declines significantly when the complexity of API specific usage increases.

5.2 Failure Analysis

To understand how the generation fails on API specifics, we conducted failure analysis on BACTX-K. The direct failure reason of the driver is collected to reveal the generation blockers. In total, 52,824 ineffective drivers were analyzed. The runtime errors of 11,095 drivers are semi-automatically analyzed while the compilation and link errors are categorized based on the compiler outputs.

Failure Taxonomies. Figure 6 details the root cause taxonomy. There are nine root causes fallen into two categories: the grammatical errors reported by compilers in build stage, and the semantic errors which are abnormal runtime behaviors identified from the short-term fuzzing results. ❶ *G1 - Corrupted Code*, the drivers do not contain a complete function of code due to either the token limitation or mismatched brackets; ❷ *G2 - Language Basics Violation*, the code violates the language basics like variable redefinition, parentheses mismatch, incomplete expressions, etc; ❸ *G3 - Non-Existing Identifier*, the code refers to non-existing things such as header files, macros, global variables, members of a `struct`, etc; ❹ *G4 - Type Error*. One main subcategory here is the code passes mismatched number of arguments to a function. The rest are either unsupported type conversions or operations such as calling non-callable object, assigning `void` to a variable, allocating an incomplete `struct`, etc; ❺ *S1 - Incorrect Input Arrangement*, the input size check either is missed when required or contains an incorrect condition; ❻ *S2 - Misinitialized Function Args*, the value or inner status of initialized argument does not fit the requirements of callee function. Typical cases are closing a file handle before passing it, using wrong enumeration value as option parameter, missing required APIs for proper initialization, etc; ❼ *S3 - Inexact Ctrl-Flow Deps*, the control-flow dependencies of a function does not properly implemented. Typical cases are missing condition checks such as ensuring a pointer is not `NULL`, missing APIs for setting up execution context, missing APIs for ignoring project internal abort, using incorrect conditions, etc. ❽ *S4 - Improper Resource Cleaning*, the cleaning API such as `xxxfree` is either missing when required or is used without proper condition checks; ❾ *S5 - Failure on Common Practices*, the code failed on standard libraries function usage like messing up memory boundary in `memcpy`, passing read-only buffer to `mkstemp`, etc. Examples of these categories are shown in [1].



(a) Number of Questions Solved by Repeat for All Configurations. **(b) Avg Pct of Solved Questions Per Repeat Round in Top-20 Configs.**

Figure 7: Statistics on the Effectiveness of Repeatedly Query.

Overall, the failures cover API usages in various dimensions: from grammatical level detail to semantic level direction, and from target API control flow conditions to dependent APIs' declarations. Improving this is challengeable since: ❶ **the involved usage is too broad to be fully put into one prompt**, which may either exceed the token limitation or distract the model; ❷ **the useful usage for generating one driver cannot be fully predetermined**. On one hand, models are inherently blackbox and probabilistics, whose mistakes cannot be fully predicted. On the other hand, there are usually multiple implementation choices for a given API.

Most failures are of mistakes in API usage specifics. The broadness of the involved usage is the major challenge.

6 Characteristics of Key Design (RQ3)

6.1 Repeatedly Query

Repeated querying is a critical aspect of prompt strategies, greatly enhancing the success rate in generating fuzz drivers regardless of employed models, temperatures, and prompt designs. Specifically, for the optimal configuration <gpt-4-0613, 0.5, ALL-ITER-K>, approximately 47.44% of the issues were resolved by reinitiating the query process (37 out of 78 total resolved issues were solved upon repetition). For the top-20 configurations, this contribution remains significantly high at an average of 67.50%.

Figure 7a displays the count of questions resolved through repeated querying across all evaluated configurations, ranked by their overall effectiveness as detailed in Table 1. This demonstrates a direct correlation between the benefit of repeated queries and the efficacy of the configuration—**the more effective a configuration, the greater the gains from repeating the queries**.

Additionally, Figure 7b presents the average percentage of questions resolved in each subsequent round of querying for the top-20 configurations. Here, the percentage for round X is determined by $\frac{Rslt(X)-Rslt(X-1)}{Rslt(1)}$, with $Rslt(X)$ indicating the number of questions resolved by round X. The X-axis starting from round two, highlighting that the first round corresponds to the initial query. This data shows that **the gain of repeated queries drops significantly after the initial few rounds**. From our evaluation, we recommend limiting repeated queries to no more than six, where the

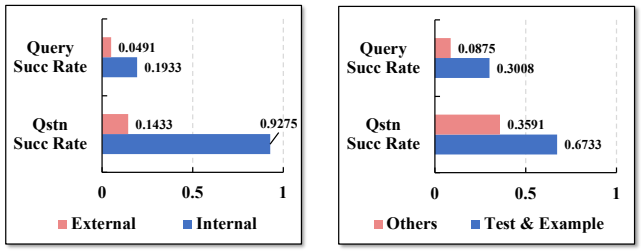


Figure 8: Average Query/Question Success Rate of Different Example Sources for All Configurations.

sixth round still manages to resolve an additional 20% of questions compared to the results of the first round.

6.2 Query With Extended Information

Querying With API Documentation. By comparing DOCTX-K and BACTX-K, we found that **there is no significant changes between their results in the metrics of resolved questions**. On one hand, a significant percentage (43%) of APIs in the evaluated questions do not have API documentation (49 out of 86 have). When there is no documentation for an API, the DOCTX-K queries are identical to BACTX-K's. On the other hand, adding API documentation in the queries may not provide enough details directly stating the API usage. This is because these API documentations usually contain a high-level description of the usage, typically a summary of main functionality with one-sentence explanations for arguments. However, the blocker-solving usage information discussed in Section 5.2, such as low level argument initialization specifics, control flow dependencies, or the usages of its dependent APIs, is usually not included.

> API documentation has minor performance benefits due to the limited usage description it contained.

Querying With Example Code Snippets. When comparing the results of BACTX-K and UGCTX-K presented in Table 1, we can clearly observe that incorporating example code snippets substantially enhances performance in most configurations. In particular, the addition of example snippets results in an average resolution of 104% more questions across the 22 evaluated configurations, which includes five models and five different temperature settings.

Nonetheless, further analysis reveals that **the inclusion of usage examples incurs a much higher token cost, with an average increase of tenfold**. The ratio of token costs for these two approaches varies from 4.20 to 39.71 across all configurations, with an average ratio of 14.65. Notably, the UGCTX-K approach demands an average of 32,367 tokens to generate a single correct solution.

Figure 8 depicts our investigation into the impact of different sources of example snippets on the quality of solutions. This figure assesses the success rates of queries/questions associated with various example sources, which are categorized in two distinct manners based on their file paths: first, as ❶ *External* vs. *Internal*, with *Internal* comprising the target project and its variations, and *External* consisting of all other sources; second, as ❷ *Test & Example* vs. *Others*, where the first group includes files with paths that contain "test" or "example" in any capitalization. The underlying data for these plots stems from questions that were solved by UGCTX-K but not by BACTX-K across all tested configurations. According to this analysis, it is clear that **both *Internal* and *Test & Example* sources are associated with significantly higher quality example snippets in comparison to their counterparts**.

Case Studies. `# 9 wc_Str_conv_with_detect` This case is challenging due to the unintuitiveness of its API usage. The API declaration is "`Str wc_Str_conv_with_detect(Str is,wc_ces * f_ces,wc_ces hint,wc_ces t_ces)`". It is used for converting the input stream `is` from one CES (character encoding scheme, `f_ces`) to another (`t_ces`). Most basic strategy drivers made mistakes on the creations of either `is` (the confusing type `Str`) or CESs, where `is` has to be created using particular APIs like `Strnew_charp_n` and CESs should be specific macros or carefully initialized `struct`. Example helps here by directly providing the usage to models.

`# 37 igraph_read_graph_graphdb` The hardest part in this case is the implicit control flow dependency it required. Besides correctly initializing the arguments, it has to call an API to mute the builtin error handlers. By default, the API will abort immediately when any abnormal input is detected, which causes frequent false crashes blocking the fuzzing progress. To mute it, the driver needs to custmoize the error handler, *e.g.*, call `igraph_set_error_handler-(igraph_error_handler_ignore)`. This requirement is hard to be inferred beforehand due to its semantic nature and few inference clues. However, some unit tests in project such as `foreign_empty.c` contain this usage, which directly instructs the generation.

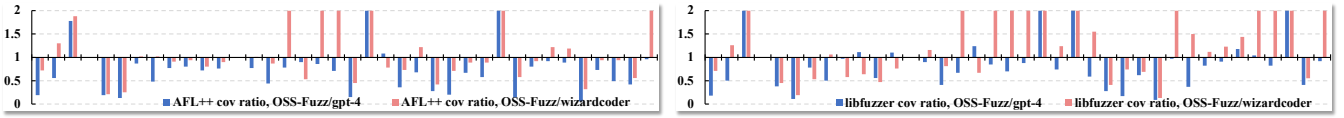
> Example code snippets can greatly enhance model performance by providing direct insights on API usage. "test/example files", "code files from the target/variant projects" are high quality sources.

6.3 Iterative Query

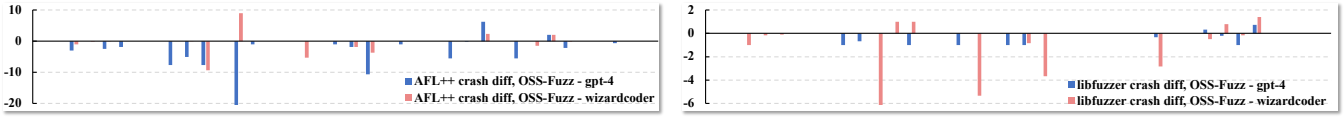
The iterative query strategy is another key design that can lead to significant improvements in performance. Referring to Table 1, we find that, on average, incorporating an iterative query strategy into BACTX-K – that is, adopting the BA-ITER-K approach – helps solve 159% more questions. Similarly, ALL-ITER-K resolves 23% more questions than UGCTX-K. However, this strategy does come at a cost. **The inclusion of iterative design tends to lead to higher token usage when generating correct solutions**. On average, the iterative strategy increases token costs by 57% for BACTX-K per successful driver generation and by 17% for UGCTX-K.

The effectiveness of the iterative strategy can be attributed to two key factors. Firstly, **it leverages a wider array of information**, including error data generated from validating previously generated drivers. Secondly, **it tackles the problem incrementally**, employing a step-by-step, divide-and-conquer approach that simplifies the complexity of the generation task. This methodology is exemplified in the case studies that follow, illustrating how the iterative strategy typically operates through practical examples.

Case Studies. `#5 md_html` This API requires the preparation of a customized callback function pointer as the argument, where all previous strategies failed to figure out. The callback function is used to handle the output data of API. All the drivers generated by `UGCTX` either pass a `NULL` pointer or a non-existing function



(a) Comparison in Average Coverage Ratio. Ratio = OSS-Fuzz/LLM, ratio < 1 → OSS-Fuzz's driver has lower coverage.



(b) Comparison in the Difference of Average Unique Crashes. Diff = OSS-Fuzz - LLM, diff < 0 → OSS-Fuzz's driver find less unique crashes.

Figure 9: Metric Comparison of LLM-Generated and OSS-Fuzz Drivers. Y-axis is the metric. For clarity, question id (x-axis) is omitted.

name. Iterative query guides the fix by providing the link error highlighting that this referred function is undefined.

#73 `pj_stun_msg_decode` This is another typical case why iterative strategy works. The initialization of its first argument has multi-level API dependencies. The dependency chain is: ❶ the API -> ❷ `pj_pool_create` -> ❸ `pj_caching_pool_init`, where -> means depends. All non-iterative strategies failed to prepare a driver with all correct usage detail of these indirect dependencies while iterative strategies solve this by providing error related feedback to LLMs and solving multiple errors one by one. In one of the solved iterative query, it first corrects the incorrect used API of ❸, then figures out the mismatched type error when calling ❷. Lastly, for the driver's runtime crash, LLMs use two rounds to fix according to the assertion code located from crash stacks.

Iterative query helps in utilizing more diverse information and solving the problem in a step-by-step manner. However, it has higher token cost and increased complexity.

7 OSS-Fuzz Driver Comparison (RQ4)

Comparison Overview. We compared LLM-generated drivers with OSS-Fuzz's to obtain more practical insights. Note that OSS-Fuzz drivers are practically used in industry for continuous fuzzing and most of them are manually written and improved for years. Particularly, LLM-generated drivers under comparison are from gpt-4-0613 and wizardcoder-15b-v1.0 using iterative strategies with temperature 0.5. These two configurations are the best representative for closed-source and open-source LLMs. In total, we evaluated 53 questions which are both resolved by all configurations. Multiple drivers of one question are merged as one to ease the comparison. This is done by adding a wrapper snippet which links the seed scheduling with the selection of the executed logic from merged drivers. Specifically, a switch structure is added to determine which driver it will execute based on a part of the input data. During each fuzzing iteration, only the logic of one merged driver is executed. Besides, some compound OSS-Fuzz drivers are designed to fuzz multiple APIs. For clear comparison, we merged all drivers of questions involved in one compound driver as one. In total, we prepared 38 drivers for each assessed LLM or OSS-Fuzz. The comparisons cover both code and fuzzing metrics such as the number of used APIs, oracles, coverage, and crashes.

Fuzzing Setup. Considering the randomness of fuzzing, we followed the suggestions from [27]: the fuzzing experiments are conducted with five times of repeat for collecting average coverage information and the fuzzing of each driver lasts for 24 hours. We used `libfuzzer` [28] and `AFL++` [18] as fuzzers with empty initial seed and dictionary. `"-close_fd_mask=3 -rss_limit_mb=2048 -timeout=30"` is used for `libfuzzer` while `AFL++`'s is the default setup of `aflpp_driver`. For fair comparison, the coverage of fuzz driver itself is excluded in post-fuzzing data collection stage (the merged driver can have thousands of lines of code) but kept in fuzzing stage for obtaining coverage feedback. In total, the experiments took 3.75 CPU year.

Code Metric: API Usage. The API usage is measured by the number of unique project APIs used in the fuzz driver. Overall, 14% (17/35) `gpt-4-0613` drivers have used less project APIs than OSS-Fuzz's while 39% for `wizardcoder-15b-v1.0`. By manually investigating these drivers, we found that **LLMs conservatively use APIs in driver generation if no explicit guidance in prompts**. For instance, some drivers only contain necessary usages such as argument initialization. And the API usage is hardly extended such as adding APIs to use an object after parsing it. This is a reasonable strategy since aggressively extending APIs increases the risk of generating invalid drivers. Adding example snippets in the prompt can alleviate this situation. As for OSS-Fuzz drivers, the API usage diversity is case by case since they are from different contributors. Some drivers, *e.g.*, [6] are minimally composed and some are extensively exploring more features of the target, *e.g.*, [10]. One interesting finding is that some OSS-Fuzz drivers are modified from the test files rather than written from scratch, which is a quite similar process as querying LLM with examples. For example, `kamailio` driver [7] is modified from test file [8]. Prompting with this example, LLM can generate similar driver code.

Code Metric: Oracle. We did statistics on the oracles of the drivers. The result is quite clear: in all 78 questions resolved by LLMs, OSS-Fuzz drivers of 15 questions contain at least one oracle which can detect semantic bugs, while there are **no LLM-generated drivers have oracles**. The used semantic oracles can be categorized as following: ❶ check whether the return value or output content of an API is expected, *e.g.*, [12]; ❷ check whether the project internal status has expected value, *e.g.*, [14]; ❸ compare whether the outputs of multiple APIs conform to specific relationships, *e.g.*, [13].

LLMs tend to generate fuzz drivers with minimal API usages, significant space are left for further improvement such as extending the use of API outputs or adding semantic oracles.

Fuzzing Metric: Coverage and Crash. Figure 9a, 9b plot the coverage and crash comparison results. Instead of presenting every

detail of the experiments for hundreds of drivers, the plots lists the comparison in certain metrics while the full experiment details can be found at [1]. Overall, **in most questions, the LLM-generated drivers demonstrate similar or better performance in metrics of both coverage and the number of uniquely found crashes**. Note that there are no false positive since the generated fuzz drivers are already filtered by the semantic checkers provided from our evaluation framework. If only the fully automatic validation process are adopted, *i.e.*, removing the last two checkers in Figure 3, the fuzzing outcome will be messed with huge number of false positives, incurring significant manual analysis efforts.

LLM-generated drivers can produce comparable fuzzing outcomes as OSS-Fuzz drivers. In large scale application, how to practically pick effective fuzz drivers is the major challenge.

8 Discussion

Relationships With OSS-Fuzz-Gen. The Google OSS-Fuzz team has undertaken a parallel work called OSS-Fuzz-Gen [49] for LLM-based fuzz driver generation. Their public information contains one security blog [56] and the source code repository [49]. Our work is complementary to theirs. Overall, at the time of submission, they put high efforts on filling the engineering gap between LLM interfaces and OSS-Fuzz projects. Their experiments are conducted on top commercial LLMs, aiming to showcase that LLM-generated fuzz drivers can help in finding zero-day vulnerabilities and reaching new testing coverage. However, there few discussion on the fundamental questions such as the design choices behind their prompt strategy, the pros and cons for different strategies, how the effectiveness varies for different models and parameters, and what are the inherent challenges and potential future directions. Our study, on the other hand, complements theirs by exploring these fundamental issues. We carefully designed prompt strategies, evaluated them on various models (open and commerical LLMs) and temperatures, and distilled findings from the results.

Contributing to OSS-Fuzz-Gen. We carefully examined the prompt strategies of OSS-Fuzz-Gen from their implementation and validated where our insights can help. Interestingly, their current strategy support part of our insights. For instance, they adopted 10 time repeat results [47] and used a lower temperature (0.4) in experiments [48]. Besides, we found that OSS-Fuzz-Gen only identifies and fixes build errors while ignoring the runtime errors caused by driver. Their generation ends when a compilable fuzz driver is synthesized and then they manually checks the validity of these drivers. To improve this, we implemented our strategies for drivers with fuzzing runtime errors in their platform, including the identification (automatic part of validation process) [50–52, 54], categorization, and the corresponding iterative fix procedure [53, 55]. These enhancements added new functionalities refining the generation results, where the cases showing its effectiveness are quickly identified [45, 46] during their benchmark tests (29 APIs, 18 projects). Currently, the improvement is merged into the main branch and is actively used to fuzz all 282 supported projects, marking a significant milestone to us. We are keeping refine our commitments, such as integrating more fine-grained error information during fix.

Potential Improvements. From our perspective, to improve the performance of LLM-based fuzz driver generation, efforts from three dimensions can be further explored. First, the domain knowledge contained inside the target scope can be modeled and utilized for better generation. For instance, to test network protocol APIs, the communication state machine of that protocol can be learned first and then used to guide the driver generation. Besides, more sophisticated prompt-based solutions can be explored, such as hybrid approaches combining traditional program analysis and prompt strategies, or agent-based approaches. Lastly, fine-tuning based methods is also a promising direction since this can enhance both the generation's effectiveness and efficiency from a model level.

Threat to Validity. One internal threat comes from the effectiveness validation of the generated drivers. To address this, we carefully examined the APIs and manually wrote tests for them to check whether the semantic constraints of a specific API have been satisfied or not. Another threat to validity comes from the fact that some OSS-Fuzz drivers, *e.g.*, code written before Sep 2021, may already be contained in the model training data, which raises a question that whether the driver is directly memorized by the model from the training data. Though it is infeasible to thoroughly prove its generation ability, which requires the retrain of LLMs, we found several evidences that supports the answers provided by these models are not memorized: Many generated drivers contain APIs that do not appear in the OSS-Fuzz drivers, especially for those drivers hinted by example usage snippets or iteratively fixed by usage and error information. Besides, the generated drivers share a distinct coding style as OSS-Fuzz drivers. For example, the generated code are commented with explanation on why the API is used and what it is used for, etc. The main external threat to validity comes from our evaluation datasets. Our study focused on C projects while the insights may not be necessarily generalizable to other languages.

9 Related Work

Fuzz Driver Generation. Several works [2, 3, 22, 24, 26, 62–64] have focused on developing automatic approaches to generate fuzz drivers. Most of these works follow a common methodology, which involves generating fuzz drivers based on the API usage existed in consumer programs, *i.e.,* programs containing code that uses these APIs. For instance, by abstracting the API usage as specific models such as trees [63], graphs [22], and automatons [62], several works propose program analysis-based methods to learn the usage models from consumer programs and conduct model-based driver synthesis. In addition, a recent work [24] emphasizes that unit tests are high quality consumer programs and proposes techniques to convert existing unit tests to fuzz drivers. Though these approaches can produce effective fuzz drivers, their heavy requirements on the quality of the consumer programs, *i.e.,* the consumers must contain complete API usage and are statically/dynamically analyzable, limit their generality. Furthermore, synthesized code often lacks human readability and maintainability, limiting their practical application. Some parallel works [32, 49] have also explored the LLM-based fuzz driver generation. However, their main goal is to build tools demonstrating the potential of LLM-based generation. Our study complements them by focusing on delivering the first comprehensive understanding of the fundamental issues in this direction.

LLM for Generative Tasks. Recent works have explored the potential of LLM models for various generative tasks, such as code

completion [57], test case generation [15, 16, 30, 41, 43, 59, 61] and code repairing [17, 39, 60]. These works utilize the natural language processing capabilities of LLM models and employ specific prompt designs to achieve their respective tasks. To further improve the models' performance, some works incorporate iterative/conversational strategies or use fine-tuning/in-context learning techniques. In test case generation, previous research works have primarily targeted on testing deep learning libraries [15, 16] and unit test generation [41, 43]. Considering the intrinsic differences between fuzz drivers and other tests and the difference on studied programming languages, these works cannot answer the fundamental effectiveness issues of LLM-based fuzz driver generation, indicating the unique values of our study.

10 Conclusion

Our study centers around answering fundamental issues of LLM-based fuzz driver generation's effectiveness. To do that, we designed a dataset and six prompt strategies, and did extensive evaluation on different models and temperatures. Our study not only established the basic understanding on this direction but also indicates the potential future improvements. Furthermore, our insights have been applied into industrial practical fuzz driver generation platform.

11 Data Availability

The source code and data involved in our study can be found at [1].

References

[1] Anonymous. 2023. Website for Our Study. https://sites.google.com/view/llm4fdg/home.

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

[3] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative Fuzzing for Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1600–1614.

[4] Codellama. 2023. codellama-34b-instruct Model. https://huggingface.co/codellama/CodeLlama-34b-hf.

[5] Open Source Contributors. 2023. Fuzz Driver of Civetweb Project. https://github.com/civetweb/civetweb/blob/master/fuzztest/fuzzmain.c.

[6] Open Source Contributors. 2023. Fuzz Driver of Coraring Project. https://github.com/RoaringBitmap/CRoaring/blob/master/fuzz/croaring_fuzzer.c.

[7] Open Source Contributors. 2023. Fuzz Driver of Kamailio Project. https://github.com/kamailio/kamailio/blob/f9cbe7ad01331e97852872c29b612409bf571c8d/misc/fuzz/fuzz_parse_msg.c#L22.

[8] Open Source Contributors. 2023. Fuzz Driver of Kamailio Project. https://github.com/kamailio/kamailio/blob/f9cbe7ad01331e97852872c29b612409bf571c8d/src/modules/misctest/misctest_mod.c#L273.

[9] Open Source Contributors. 2023. Fuzz Driver of Libmodbus Project. https://github.com/google/oss-fuzz/blob/master/projects/libmodbus/fuzz/FuzzClient.c.

[10] Open Source Contributors. 2023. Fuzz Driver of Lua Project. https://github.com/google/oss-fuzz/blob/master/projects/lua/fuzz_lua.c.

[11] Open Source Contributors. 2023. Fuzz Driver of Tmux Project. https://github.com/tmux/tmux/blob/master/fuzz/input-fuzzer.c.

[12] Open Source Contributors. 2023. The PermaLink for Oracle Checking Expected Output of One API Used in Fuzz Driver of Bind9 Project. https://gitlab.isc.org/isc-projects/bind9/-/blob/af5d0a0afbbc136074133e80a63565d668b8a40d/fuzz/dns_rdata_fromwire_text.c#L176.

[13] Open Source Contributors. 2023. The PermaLink for Oracle Checking Expected Relationship of Two APIs' Outputs Used in Fuzz Driver of Bind9 Project. https://gitlab.isc.org/isc-projects/bind9/-/blob/af5d0a0afbbc136074133e80a63565d668b8a40d/fuzz/dns_name_fromwire.c#L82.

[14] Open Source Contributors. 2023. The PermaLink for Oracle Used in Fuzz Driver of iGraph Project. https://github.com/igraph/igraph/blob/1b25b1102916bb99274af4bc7c6322f2fe193204/fuzzing/read_dl_fuzzer.cpp#L62.

[15] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. arXiv:2212.14834 [cs.SE]

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. *arXiv preprint arXiv:2304.02014* (2023).

[17] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. *arXiv preprint arXiv:2205.10583* (2022).

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[19] Google. 2017. OSS-Fuzz Github Repository. https://github.com/google/oss-fuzz.

[20] Google. 2023. Google PaLM2 Text Models. https://cloud.google.com/vertex-ai/docs/generative-ai/model-reference/text.

[21] John Hancock. 2004. *Jaccard Distance (Jaccard Index, Jaccard Similarity Coefficient)*. https://doi.org/10.1002/9780471650126.dob0956

[22] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*.

[23] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[24] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. Hwang. 2023. UTOPIA: Automatic Generation of Fuzz Driver using Unit Tests. In *2023 2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 746–762. https://doi.org/10.1109/SP46215.2023.00043

[25] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. arXiv:2303.06689 [cs.SE]

[26] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE: fuzzing Windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[28] libfuzzer@googlegroups.com. 2019. LibFuzzer. https://llvm.org/docs/LibFuzzer.html.

[29] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[30] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. *arXiv preprint arXiv:2212.04732* (2022).

[31] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568* (2023).

[32] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation. arXiv:2312.17677 [cs.CR]

[33] OpenAI. 2022. ChatGpt Website. https://chat.openai.com.

[34] OPENAI. 2023. Example Usage of System Role. https://platform.openai.com/docs/guides/text-generation/chat-completions-api.

[35] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[36] OpenAI. 2023. OpenAI GPT-3.5 Models List. https://platform.openai.com/docs/models/gpt-3-5.

[37] OpenAI. 2023. OpenAI GPT-4 Models List. https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo.

[38] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.

[39] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. 2023. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–18. https://doi.org/10.1109/SP46215.2023.00001

[40] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[41] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).

[42] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive Test Generation Using a Large Language Model. arXiv:2302.06527 [cs.SE]

[43] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. *arXiv preprint arXiv:2305.00418* (2023).

[44] Sourcegraph. 2024. Sourcegraph CLI tool. https://docs.sourcegraph.com/cli.

[45] Google OSS-Fuzz Team. 2024. Feedback of adding runtime error filtering. https://github.com/google/oss-fuzz-gen/pull/185#issuecomment-2024141597.

[46] Google OSS-Fuzz Team. 2024. Observed case for successfully generating fuzz drivers for new APIs by fixing the fuzzing runtime error. https://github.com/google/oss-fuzz-gen/pull/198#issuecomment-2044547163.

[47] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen Default Repeat Time in Experiments. https://github.com/google/oss-fuzz-gen/blob/d2b1bfac45efe217b28d5d882d3c3b942adaf9f7/report/docker_run.sh#L111.

[48] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen Default Temperature in Experiment. https://github.com/google/oss-fuzz-gen/blob/d2b1bfac45efe217b28d5d882d3c3b942adaf9f7/run_one_experiment.py#L53.

[49] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen: LLM powered fuzzing via OSS-Fuzz. https://github.com/google/oss-fuzz-gen.

[50] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen PR #185. https://github.com/google/oss-fuzz-gen/pull/185.

[51] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen PR #187. https://github.com/google/oss-fuzz-gen/pull/187.

[52] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen PR #191. https://github.com/google/oss-fuzz-gen/pull/191.

[53] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen PR #198. https://github.com/google/oss-fuzz-gen/pull/198.

[54] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen PR #199. https://github.com/google/oss-fuzz-gen/pull/199.

[55] Google OSS-Fuzz Team. 2024. OSS-Fuzz-Gen PR #204. https://github.com/google/oss-fuzz-gen/pull/204.

[56] Google OSS-Fuzz Team. 2024. Scaling security with AI: from detection to solution. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html.

[57] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source Code Is All You Need. *arXiv preprint arXiv:2312.02120* (2023).

[58] WizardLM. 2023. wizardcoder-15b-v1.0 Model. https://huggingface.co/WizardLM/WizardCoder-15B-V1.0.

[59] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[60] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[61] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. White-box compiler fuzzing empowered by large language models. *arXiv preprint arXiv:2310.15991* (2023).

[62] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. 2023. Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation. (2023).

[63] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. 2811–2828.

[64] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 318–327.