When Fuzzing Meets LLMs: Challenges and Opportunities

Yu Jiang
Tsinghua University

Jie Liang
Tsinghua University

Fuchen Ma
Tsinghua University

Yuanliang Chen
Tsinghua University

Chijin Zhou
Tsinghua University

Yuheng Shen
Tsinghua University

Zhiyong Wu
Tsinghua University

Jingzhou Fu
Tsinghua University

Mingzhe Wang
Tsinghua University

Shanshan Li
NUDT

Quan Zhang
Tsinghua University

ABSTRACT

Fuzzing, a widely-used technique for bug detection, has seen advancements through Large Language Models (LLMs). Despite their potential, LLMs face specific challenges in fuzzing. In this paper, we identified five major challenges of LLM-assisted fuzzing. To support our findings, we revisited the most recent papers from top-tier conferences, confirming that these challenges are widespread. As a remedy, we propose some actionable recommendations to help improve applying LLM in Fuzzing and conduct preliminary evaluations on DBMS fuzzing. The results demonstrate that our recommendations effectively address the identified challenges.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Large Language Model, Fuzzing

ACM Reference Format:
Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. 2024. When Fuzzing Meets LLMs: Challenges and Opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663784

1 INTRODUCTION

Fuzzing is a promising technique for software bug detection [8, 30, 37, 40, 47]. Large Language Models (LLM) are rapidly gaining popularity across various applications for their versatility and capability [17, 18]. From natural language processing [7, 25, 29] to code generation [22, 27], LLM's broad utility is making it a prominent and sought-after solution in diverse domains. This development has naturally influenced fuzzing research: to help improve the fuzzing effectiveness, LLM has now become one of the key enablers to assist the core processes of fuzzing, including driver synthesis [31, 44, 45], input generation [10, 11, 46], and bug detection [9, 12, 20].

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663784

While excelling in natural language analysis, LLM encounters some common pitfalls like limited context length [23] and hallucination problems [19, 26, 34], etc. Consequently, LLM exhibits limitations in complex program analysis. These pitfalls of LLM affect the effectiveness of fuzzing, leading to testing performance degradation, manifesting as high false positives, low test coverage, and limited scalability.

In this paper, we identify five common challenges when using LLM-based fuzzing technology: 1) Firstly, they often produce low-quality outputs in fuzzing driver synthesis, lacking the precision required for effective bug detection. 2) Secondly, these models demonstrate a limited scope in their understanding and processing capabilities, constraining their utility in diverse fuzzing scenarios. 3) Thirdly, LLMs struggle with generating sufficiently diverse inputs during the fuzzing process, which is critical for thorough and effective bug detection. 4) Fourthly, they face challenges in maintaining the validity of generated inputs, a crucial factor for accurate and reliable fuzzing. 5) Lastly, LLMs' inaccurate understanding of bug detection mechanisms hinders their ability to identify and address complex software vulnerabilities effectively, thereby limiting their overall effectiveness in the fuzzing process. We performed a comprehensive survey and revisited most recent fuzzing works that rely on LLM for tackling different problems in the fuzzing process. To our surprise, the results show that each work encounters at least one of these challenges.

Although LLMs are widespread, it is more important for us to avoid its weakness, and at the same time take advantage of its strengths. To this end, we perform an impact analysis of the implications in three key fuzzing steps. These findings inspire us with some opportunities for better usage of LLM in each fuzzing step according to whether the corresponding corpus and documentation are rich. Furthermore, we performed some preliminary evaluations according to these opportunities by applying LLM in fuzzing database management systems(DBMS). The results demonstrate that the reasonable instantiation of those recommendations can overcome the challenges in LLM-assisted DBMS fuzzing.

2 CHALLENGES AND OPPORTUNITIES

Despite that LLM have achieved great success, the application of LLM in fuzzing is often prone to several problems, ranging from deduction accuracy to adapt scalability. Overlooking these issues may result in poor seed quality or omitting critical bugs, leading to a limited fuzzing performance. In this section, we summarize the five challenges that commonly occur when applying LLM in fuzzing. While these challenges might initially appear straightforward, they usually stem from small shortcomings that are typical in fuzzing. We group these challenges with respect to the states of a typical

fuzzing workflow, as depicted in Figure 1.

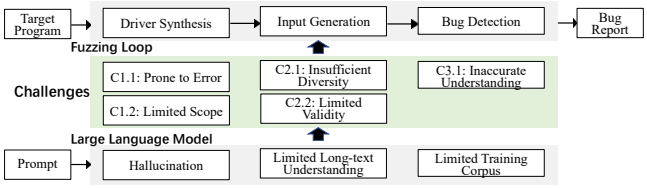

Figure 1: Fuzzing Workflow with LLM enhanced.

2.1 Driver Synthesis

Description. Recently, several pioneer works have been proposed to utilize LLMs to enhance driver synthesis [12, 13, 31, 43, 44]. Their basic idea is to use API documentation as the prompt context, and then ask LLMs to generate API invoking sequences as fuzzing drivers. For example, both TitanFuzz [12] and PromptFuzz [31] design customized prompt templates to guide LLMs in generating code that follows programming syntax and semantics.

Challenges. The application of LLMs to driver synthesis can be ineffective if done directly, as LLMs have a tendency to produce hallucinations [7, 23] and perform less effectively on programs that are not included in their training corpus [23]. These limitations present two challenges for driver synthesis. The first one is that the synthesized drivers are *prone to error*, leading to a non-negligible number of false positives during fuzzing. For example, according to comprehensive evaluation on LLM-based driver synthesis for OSS-Fuzz projects [44], GPT-4 can correctly generate roughly 40% drivers, while the rest of the drivers contain errors. Among the erroneous drivers, 93% exhibit one or more of the following issues: type errors, mis-initialized function arguments, usage of non-existing identifiers, and imprecise control-flow dependencies. This occurrence primarily arises due to LLMs relying on pre-trained knowledge for driver synthesis, leading to the production of hallucinations [19]. The second challenge is that the application of directly using LLMs for driver synthesis has *limited scope* because LLMs have limited knowledge on unseen programs. For those target programs, LLMs sometimes use training knowledge to fill the gap, thus generating incorrect API invoking sequences. For example, developers from Google's OSS-Fuzz project [39] attempted to leverage LLMs to synthesize drivers. Out of 31 tested OSS-Fuzz projects, 14 successfully compiled new targets and increased coverage with the synthesized drivers. The drivers unsuccessfully synthesized by LLMs typically originated from less common projects like `krb5` and `rtpproxy`. In contrast, LLMs are more likely to generate compilable and effective drivers for more common projects, such as `tinyxml2` and `cjson`.

Recommendations. We have the following recommendations:

`REC 1.1` Some targets whose code or use cases have been included in the training corpus. *For these cases, employing LLM for automated synthesis of fuzz drivers, complemented by error-guided corrective measures, is a practical approach.* Iteratively querying the LLM based on identified errors and fixing the errors are practical measures [44], which helps to address the *prone-to-error* challenge.

For example, `libpng` is a common library and has already been seen by GPT4 in its training process. Consequently, it is possible to directly ask GPT4 to generate a fuzz testing driver for `libpng` by giving the prompt "Generating LLVMFuzzerTestOneInput for test libpng." However, the generated driver might still contain errors in grammar or encounter issues during the process of compiling and linking. Test engineers can subsequently submit individual LLM queries containing the error messages to rectify these issues, occasionally necessitating multiple iterations.

`REC 1.2` For targets without a dedicated corpus in training, one can collect valuable materials such as function prototypes, example programs, or connection rules between functions. *Conducting prompt engineering which involves embedding these materials, enhances the precision in generating logical sequences of function calls for the creation of drivers.* The prompt engineering approach is a practical solution to tackle the challenge of *limited scope*.

For example, `typst` is a new markup-based typesetting system like LaTex and claims it is more easier to learn and use. To generate a fuzz driver for it, feed the prompt "Generate LLVMFuzzerTestOneInput for typst" to ChatGPT-3.5 will encounter hallucination problems and generate a completely non-existent driver. Instead, the project `typst` has lots of documents and unit tests. Feeding these materials that illustrate the usage of the functions is helpful for LLMs to generate effective drivers [39]. Additionally, it is also feasible to iteratively query LLMs to address any errors that may be present in the drivers.

`REC 1.3` Sometimes, even with adequate documentation and examples, LLMs can still encounter challenges in generating valid drivers at times, especially for extremely complex targets like Linux kernel. These systems frequently involve intricate dependencies among their APIs, or there exist implicit dependencies among lower-level systems that pose challenges for LLM to capture. *For these targets, it is advisable to refrain from relying on LLMs. Instead, it is more practical and feasible to explore conventional methods.*

For example, KSG [36] uses the ebpf to dynamically infer the kernel's system call argument type and value constraints, and successfully generate 2,433 Syzlang Spec. In contrast, LLM-based approaches use static inference based on kernel man pages and source code. But they may find some complex dummy operations. And it's hard for them to deduct pointer references.

2.2 Input Generation

Description. Recently, several pioneer works [5, 38, 41, 42] have been proposed to utilize LLM to enhance input generation. Their basic idea is to use input specifications and input examples as the prompt context and then ask LLMs to generate new inputs. For example, LLMFuzzer [5] feeds input specifications to LLMs to generate initial seeds for mutation-based fuzzers.

Challenges. The application of LLMs to input generation can be ineffective if done directly, as LLMs heavily rely on training corpus and have limited long-text understanding [23, 35]. These limitations present two challenges for input generation. The first one is that the generated inputs have *insufficient diversity*, leading to inefficient exploration of the input space. This is because LLMs are pre-trained models and prone to responding to users' queries in a similar manner when given the same prompt context. Therefore, it is difficult for LLMs to generate diverse inputs if they only provide limited information. For example, when applying ChatAFL [32] to the RTPS protocol fuzzing directly, if only a limited amount of protocol information is provided in the prompts, LLMs can only generate inputs that cover 4 states out of 10 states that the RTPS protocol supported. This results in a substantial portion of the RTSP state remaining unexplored. The second challenge is that the generated inputs often have *limited validity*, leading to early termination when the target program executes these inputs. This is

because LLMs cannot fully understand the long texts of input formats or examples due to limited ability on long text processing [35]. For example, Border Gateway Protocol (BGP) is a complex protocol, whose document (BGP RFC 9952) has more than 28,000 words to describe its functionalities. When generating inputs of BGP based on the RFC description, LLMs usually forget to generate the length field of the TLV substructures in the BGP message because the description of the main message structure and the TLV substructures are a little far, making LLMs hard to totally understand BGP format.

Recommendations. We have the following recommendations:

REC 2.1 Some of the testing inputs to the system are common and have a large number of examples on the web, and they have been included in the LLM's training corpus. *It is possible to directly employ LLM to generate test cases for them, combining methodologies focused on diversification.* These methods encompass internal approaches, such as meticulously crafted prompts that demand using diverse features, as well as external methods, such as coverage-guided genetic algorithms. They both contribute to address the challenge of *insufficient diversity*.

For instance, when testing common text protocols such as HTTP and FTP, where LLM excels in its support for text-based languages, it is feasible to directly instruct LLM to generate test cases for these protocols. To increase diversity, for internal approaches, we can use prompts that encourage LLM to generate HTTP files with various methods (e.g., GET, POST, PUT), different headers, different query parameters, URL structures, various payloads, and other aspects. We can also interactively ask LLM to cover more types of messages [32]. For external approaches, we can utilize coverage-guided generation used in conventional fuzzing along with more real-world examples to enhance LLM.

REC 2.2 In many cases, the LLM is not trained with a dedicated training corpus specifically tailored for the test subjects. *Rather than employing LLM directly for generating the final test cases, we suggest utilizing LLM to transform well-known knowledge to formulate the input specifications or build initial test cases.* The input specification helps address the challenge of *limited validity*, and the initial test cases help address the challenge of *insufficient diversity*.

For instance, in the case of protocol implementations lacking machine-readable grammar, generating valid test inputs automatically to adhere to the necessary structure and order becomes challenging. In such scenarios, leveraging that LLM has been trained on established protocols, allows the transfer of grammars from these protocols with the assistance of LLM and recorded message sequences. The grammar can enhance the validity of the generated test cases. With the grammar, conventional grammar-based fuzzers could be utilized to generate more test cases [15, 16, 32]. Another instance is transforming test cases of popular database systems to initial seeds for the tested database system. The SQL queries of popular database systems like PostgreSQL have rich diversity and they have already been trained for LLM. Therefore, leveraging the knowledge of LLM to transform them into the format of the target database system is feasible. Providing them to the fuzzer as the initial seed helps enhance the diversity of generated test cases.

2.3 Bug Detection

Description. Recently, several pioneer works [24, 28] utilize LLM to enhance bug detection. Their basic idea is to use functionality descriptions of the target program as the prompt context, and then ask LLMs to generate code that implements the same functionalities with the target program. By comparing the execution results of the two functionally equivalent programs, they can detect logic bugs in the target program. For example, Differential Prompting [28] queries LLMs about the intention of a piece of provided code and then uses the obtained intention as a new prompt context for LLMs to generate code with the same intention.

Challenges. The application of LLMs to bug detection can be ineffective if done directly, as LLMs have limited long-text understanding [35], posing a challenge to *inaccurate understand* of the semantics of the target program. For example, researchers [28] found that LLMs may misconstrue code designed to identify the longest common substring as being intended for finding the longest common subsequence. This misinterpretation can occur even though these two problems require entirely distinct code solutions. As a result, LLMs may generate code whose functionality deviates from the target program, thus leading to an inaccurate test oracle. According to the experiment results of Differential Prompting [28], it achieves 66.7% success rate when generating reference implementation for programs from the programming contest website Codeforces. While this is substantially better than its baseline, it still results in a false-positive rate of 33.3%, which is still not sufficient for practical usage.

Recommendations. We have the following recommendations:

REC 3.1 Defining test oracles is highly dependent on specific targets and scenarios, presenting the most formidable aspect of fuzzing. *For complicated targets, we suggest to avoid analyzing results with LLM directly. Instead, consider employing LLM to extract features or patterns associated with a specific bug type, leveraging domain knowledge.* Subsequently, monitoring the system using these patterns aids in addressing the challenge of *inaccurate understanding*.

For example, many time-series databases like IoTDB implicitly handle exceptions. Consequently, the system will not crash or exhibit other abnormal behaviors. Nevertheless, these database systems generate extensive logs, and errors manifest as exceptions in these logs. Therefore, it becomes feasible to use LLM for analyzing the logs to discern error patterns. In such scenarios, we recommend employing LLM to scrutinize the logs, identify error patterns, and subsequently leverage these patterns for detecting logic errors.

REC 3.2 Some targets or projects contain well-defined documentations, where the expected behaviors are clearly described, like the RFCs for protocols. *For these cases, we suggest to leverage the natural language understanding ability of LLM to extract the expected behaviors from the documentations for test oracle definition.* This helps LLM to understand the intention and design of the target programs, thus addressing the challenge of *inaccurate understanding*.

For example, the RFCs for protocols usually contain detailed descriptions of the protocol's expected behaviors. Take the RFC 854 [4] for Telnet protocol as an example. It specifies expected behaviors during the negotiation of some disabled command options or unnegotiated commands. These can be used as test oracles and can be further used to uncover CVE-2021-40523 [33].

3 POTENTIAL SOLUTIONS

To demonstrate the practicality of our recommendations, we use the Database Management System (DBMS) as the target for LLM-assisted fuzzing. Addressing challenges in driver synthesis, input generation, and bug detection, we propose three potential solutions: state-aware driver synthesis, cross-DBMS SQL transfer, and log-based Oracle definition. These solutions are compared with rudimentary uses of LLM, where it is directly employed.

3.1 LLM-Enhanced Connector Synthesis

Obstacle: Database connectors or database drivers link applications to databases via defined interfaces including functions and parameters. Fuzzing drivers consist of these interface sequences. Directly using LLM to generate database drivers faces two challenges. First is *prone to error*: API sequences hold semantic details within the connector's state, and directly generating sequences may import errors. Second is *limited scope*: LLM lacks the state transition knowledge of the connectors due to limited training data.

Solution: Following REC 1.2 , we propose LLM-enhanced state-aware database connector synthesis. We first collect JDBC function prototypes and example programs that utilize JDBC. Then we model the connection relationships between JDBC functions as state-transition rules. Next, we gather the function prototypes, example programs, and connection rules as input for LLM. The prompt we give is like " Based on the state-transition rules and state description of functions, please generate a sequence of APIS within length 15. It is required to cover a different combination of state transitions than before."

Result: We implement LLM-enhanced connector synthesis into WINGFUZZ^{conn} and compare it against LLM^{conn}, which directly utilizes LLM to generate drivers for MySQL Connector/J [3], MariaDB Connector/J [2], and AWS JDBC Driver for MySQL [1]. We perform fuzzing on ClickHouse for each tool. Table 1 shows the driver correctness ratios and branch coverage by LLM^{conn} and WINGFUZZ^{conn} on three selected DBMSs in 12 hours. These statistics show that WINGFUZZ^{conn} always performs better in both driver correctness ratio and branch coverage than LLM^{conn}. The main reason is that the state-transition rules embed semantic information, and it also helps LLM generate API sequences that account for the diverse states within the database connector.

Table 1: Driver Correctness Ratios and Branch Coverage.

DBMS	Driver Correctness Ratios		Branch Coverage	
	LLM^{conn}	WINGFUZZ^{conn}	LLM^{conn}	WINGFUZZ^{conn}
MariaDB Connector/J	0.142	0.331	583	843
MySQL Connector/J	0.216	0.367	1256	1982
AWS MySQL JDBC	0.203	0.394	1382	2293

3.2 Cross-DBMS SQL Transfer

Obstacle: SQL queries, as the inputs of DBMS, are vital to DBMS fuzzing. Generating SQL queries directly via LLM faces two main challenges: ensuring semantic correctness and promoting query diversity. The intricate SQL grammar, encompassing various clauses, expressions, and rules, poses a challenge for LLM in achieving semantic correct, which is vital for triggering complex DBMS behaviors. Furthermore, diversity in SQL queries is crucial for probing deep DBMS logic. However, LLM's constrained variety limits the exploration of diverse query structures.

Solution: We introduce the cross-DBMS SQL transfer approach, following recommendation REC 2.2 , for SQL generation. Instead of directly creating SQL queries, we utilize LLM to transfer test cases from other DBMSs as initial seeds to fuzz the target DBMS. It contains three steps. First, executing existing test cases in their native DBMS to capture schema information; second, feeding schema information to LLMs to generate new test cases; third, temporarily commenting out unparsable sections, ensuring proper parsing, and subsequently uncommenting them after mutation.

Result: We implement the solution called WINGFUZZ^{input} and compare it with LLM^{input}, which directly uses LLM to generate the SQL queries. We run two tools on three DBMS: MonetDB [6], DuckDB [14], and ClickHouse [21]. Table 2 shows semantic correctness ratios and covered branches of two tools on three DBMSs in 12 hours. It is found that WINGFUZZ^{input} performs better than LLM^{input} on DBMS fuzzing. Specifically, the test cases generated by WINGFUZZ^{input} contain 159.35%, 36.65%, and 112.14% more semantic-correct SQL statements, and cover 55.96%, 21.83%, and 16.41% more branches than that of LLM^{input} across three DBMS, respectively. It indicates that LLM cannot directly generate high-quality SQL queries for DBMS fuzzing. The main reason is that the transfer seeds improve the diversity of mutated test cases, and the fuzzer's mutator promises the semantic correctness of SQL queries.

Table 2: Semantic Correctness Ratios and Branch Coverage.

DBMS	Semantic Correctness Ratios		Branch Coverage	
	LLM^{input}	WINGFUZZ^{input}	LLM^{input}	WINGFUZZ^{input}
MonetDB	0.1594	0.4134	26,828	41,840
DuckDB	0.2551	0.3486	57,937	70,583
ClickHouse	0.1458	0.3093	124,887	145,383

3.3 Monitor-Based DBMS Bug Detection

Obstacle: The most critical step for DBMS bug detection is to construct the test oracles to identify the logic or performance bugs. A test oracle determines the correctness or validity of the DBMS's behaviors. Directly using LLMs to construct test oracles is challenging as LLMs lack specific knowledge about DBMS's behaviors.

Solution: To address the challenges, we propose the Runtime Monitor-Based DBMS Bug Detection following the REC 3.1 , which detects the anomalies of DBMS by analyzing the runtime information of DBMS. DBMS usually contains the implicit exception handler mechanism to avoid system crashes, which usually output key internal states of DBMS. Unlike constructing the oracle by checking the execution result of the SQL query, our approach involves using LLM to analyze the runtime information for bug detection. The process contains two main steps. First, it instruments an agent to extract the runtime information of DBMS. Then, it uses LLM to detect the anomaly with predefined error patterns.

Table 3: Number of Reported Bugs and Real Bugs.

DBMS	LLM^{bug}		WINGFUZZ^{bug}	
Name	Reported	Real	Reported	Real
MonetDB	61	0	6	3
DuckDB	54	0	5	3
ClickHouse	67	1	3	3

Result: To evaluate the effectiveness of our recommendation, we implement the solution with WINGFUZZ^{bug} and compare it with LLM^{bug}, which directly uses LLM to determine whether the execution of the SQL query is right during the fuzz loop. Table 3 shows the number of reported bugs and real bugs by two tools in 12 hours on MonetDB, DuckDB and ClickHouse. It shows WINGFUZZ^{bug} can detect more anomalies with fewer false positives than LLM^{bug}. It is because that the runtime information contains the error message of DBMS, which helps LLM to analyze and detect bugs.

4 CONCLUSION

We systematically analyze five challenges when using LLM in fuzzing and confirm their prevalence through a review of recent top-tier conference papers. These challenges affect the effectiveness of the LLM-based fuzzing technologies. To address them, we provide recommendations to assist the main steps in fuzzing, which have demonstrated effectiveness in our preliminary experiments.

REFERENCES

[1] 2023. aws-mysql-jdbc. https://github.com/awslabs/aws-mysql-jdbc. Accessed: May 6, 2024.

[2] 2023. mariadb-connector-j. https://github.com/mariadb-corporation/mariadb-connector-j. Accessed: May 6, 2024.

[3] 2023. mysql-connector-j. https://github.com/mysql/mysql-connector-j. Accessed: May 6, 2024.

[4] 2023. Rfc854. https://datatracker.ietf.org/doc/html/rfc854. Accessed: May 6, 2024.

[5] Joshua Ackerman and George Cybenko. 2023. Large Language Models for Fuzzing Parsers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop*. 31–38.

[6] MonetDB B.V. 2023. MonetDB Website. https://www.monetdb.org. Accessed: May 6, 2024.

[7] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Kaijie Zhu, Hao Chen, Linyi Yang, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109* (2023).

[8] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, 1967–1983.

[9] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. 2023. Tyr: Finding Consensus Failure Bugs in Blockchain System with Behaviour Divergent Model. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2517–2532. https://doi.org/10.1109/SP46215.2023.10179386

[10] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2023. Effective test generation using pre-trained large language models and mutation testing. *arXiv preprint arXiv:2308.16557* (2023).

[11] Victor Dantas. 2023. Large Language Model Powered Test Case Generation for Software Applications. (2023).

[12] 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. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[13] 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. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[14] DuckDB. 2023. DuckDB WebSite. https://www.duckdb.org/. Accessed: May 6, 2024.

[15] J. Fu, J. Liang, Z. Wu, and Y. Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1799–1810. https://doi.ieeecomputersociety.org/

[16] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2023. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. New York, NY, USA, Article 49, 12 pages. https://doi.org/10.1145/3551349.3560431

[17] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. *Authorea Preprints* (2023).

[18] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620* (2023).

[19] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232* (2023).

[20] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2023. Automated Bug Generation in the era of Large Language Models. *arXiv preprint arXiv:2310.02407* (2023).

[21] ClickHouse Inc. 2023. ClickHouse Website. https://clickhouse.com. Accessed: May 6, 2024.

[22] Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2023. Benchmarking and Explaining Large Language Model-based Code Generation: A Causality-Centric Approach. *arXiv preprint arXiv:2310.06680* (2023).

[23] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. 2023. Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169* (2023).

[24] Siva Kesava Reddy Kakarla and Ryan Beckett. 2023. Oracle-based Protocol Testing with Eywa. *arXiv preprint arXiv:2312.06875* (2023).

[25] Prateek Kumar and Sanjay Kathuria. 2023. Large language models (LLMs) for natural language processing (NLP) of oil and gas drilling data. In *SPE Annual Technical Conference and Exhibition?* SPE, D021S012R004.

[26] Katherine Lee, Orhan Firat, Ashish Agarwal, Clara Fannjiang, and David Sussillo. 2018. Hallucinations in neural machine translation. (2018).

[27] Jia Li, Ge Li, Chongyang Tao, Huangzhao Zhang, Fang Liu, and Zhi Jin. 2023. Large Language Model-Aware In-Context Learning for Code Generation. *arXiv preprint arXiv:2310.09748* (2023).

[28] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 14–26.

[29] Zhengliang Liu, Tianyang Zhong, Yiwei Li, Yutong Zhang, Yi Pan, Zihao Zhao, Peixin Dong, Chao Cao, Yuxiao Liu, Peng Shu, et al. 2023. Evaluating large language models for radiology natural language processing. *arXiv preprint arXiv:2307.13693* (2023).

[30] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA, 4481–4498.

[31] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation. *arXiv preprint arXiv:2312.17677* (2023).

[32] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[33] MITRE. 2021. CVE-2021-40523. (2021).

[34] Anna Rohrbach, Lisa Anne Hendricks, Kaylee Burns, Trevor Darrell, and Kate Saenko. 2018. Object hallucination in image captioning. *arXiv preprint arXiv:1809.02156* (2018).

[35] Uri Shaham, Maor Ivgi, Avia Efrat, Jonathan Berant, and Omer Levy. 2023. ZeroSCROLLS: A Zero-Shot Benchmark for Long Text Understanding. *arXiv preprint arXiv:2305.14196* (2023).

[36] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. {KSG}: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 351–366.

[37] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[38] Elwin Tamminga, Bouwko van der Meijs, and Ultraware Stjepan Picek. 2023. Utilizing Large Language Models for Fuzzing: A Novel Deep Learning Approach to Seed Generation. (2023).

[39] Google Open Source Security Team. [n. d.]. AI-Powered Fuzzing: Breaking the Bug Hunting Barrier. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html. Accessed: May 6, 2024.

[40] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice* (Virtual Event, Spain) *(ICSE-SEIP '21)*. 328–337. https://doi.org/10.1109/ICSE-SEIP52600.2021.00042

[41] 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).

[42] 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).

[43] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. *arXiv preprint arXiv:2401.00563* (2023).

[44] Cen Zhang, Mingqiang Bai, Yaowen Zheng, Yeting Li, Xiaofei Xie, Yuekang Li, Wei Ma, Limin Sun, and Yang Liu. 2023. Understanding large language model based fuzz driver generation. *arXiv preprint arXiv:2307.12469* (2023).

[45] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. 2023. Daisy: Effective Fuzz Driver Synthesis with Object Usage Sequence Analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 87–98. https://doi.org/10.1109/ICSE-SEIP58684.2023.00013

[46] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. 2023. Towards Better Semantics Exploration for Browser Fuzzing. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 243 (oct 2023), 28 pages. https://doi.org/10.1145/3622819

[47] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. New York, NY, USA, 1135–1147. https://doi.org/10.1145/3540250.3549107

Received 20-JAN-2024; accepted 2024-04-09