Large Language Models Based Fuzzing Techniques: A Survey

Linghan Huang[1] , **Peizhou Zhao**[1] , **Huaming Chen**[1] , **Lei Ma**[2,3]
[1]University of Sydney
[2]The University of Tokyo
[3]University of Alberta

Abstract

In the modern era where software plays a pivotal role, software security and vulnerability analysis have become essential for software development. Fuzzing test, as an efficient software testing method, are widely used in various domains. Moreover, the rapid development of Large Language Models (LLMs) have facilitated their application in the field of software testing, demonstrating remarkable performance. Considering that existing fuzzing test techniques are not entirely automated and software vulnerabilities continue to evolve, threr is a growing trend towards employing fuzzing test generated based on large language models. This survey provides a systematic overview of the approaches that fuse LLMs and fuzzing tests for software testing. In this paper, a statistical analysis and discussion of the literature in three areas, namely LLMs, fuzzing test, and fuzzing test generated based on LLMs, are conducted by summarising the state-of-the-art methods up until 2024. Our survey also investigate the potential for widespread deployment and application of fuzzing test techniques generated by LLMs in the future.

1 Introduction

Large language models have presented powerful performance in various fields, include the field of software testing. Software testing generated based on large language models shows improvement in terms of efficiency and accuracy compared to traditional software testing systems. Automated software testing is becoming a significant highlight. Fuzzing test has been widely employed since the late 19th century. Its principle involves generating a series of unexpected inputs to test the reliability and security of software. With the development of the software field, fuzzing test holds an important position in software testing. The technology of generating fuzzing test based on large language models (LLMs-based fuzzers) for automated software testing is an innovative topic. By the end of 2023, such techniques have already emerged. Examples include TitanFuzz [Deng *et al.*, 2023a], FuzzGpt [Deng *et al.*, 2023b], and other fuzzers targeting different software types. They combine different large language models with fuzzing test techniques to develop new fuzzing test systems. The following will further discuss this field.

1.1 Survey contribution

In this survey, we explored the answers to three research questions in the field of LLMs-based fuzzers. Here are the three research questions:

1. How is llm used for fuzzing test in the AI and non-AI fields?(Subsection 4.1 and 4.2)
2. Compared with traditional fuzzers, what are the advantages of LLMs-based fuzzer?(Subsection 4.1 and 4,2)
3. What is the future development trend of LLMs-based fuzzer?(Section 5)

We created a Github repository, it contains the core literature we collected and used in this paper. Mainly about the field of LLMs-based fuzzers. The link as below: https://github.com/EdPuth/LLMs-based-Fuzzer-Survey

1.2 Survey Scope and literature quality assessment

We select 14 core literature through predetermined criteria, manual screening, and snowballing methods. Our literature review is mainly dedicated to explaining and analyzing all literature on the combined application of fuzzing test and large language models. When collecting papers, we will refer to the following selection criteria. If a paper meets any of these criteria, it will be included.

- The literature proposes or improves a testing tool, framework, and method for fuzzing test based on the LLM concept.
- Literature research involves testing methods, concepts, and frameworks of fuzzing test in LLM environment.
- Discuss the relationship between fuzzing test and LLM concepts, including theoretical research and methodological discussions.
- Discuss the advantages, potential limitations, and future analysis of the combined application of LLM and fuzzing test.

2 Background

2.1 Large Language Model (LLM)

The emergence of large language models has provided great help for different complex language tasks, such as translation, summary, information retrieval, dialogue interaction, etc [Naveed *et al.*, 2023]. The reason why the large language model is so powerful is because the transformers mechanism is added to the model's framework, which greatly improves the model's computing power.

According to statistics by Humza Naveed, Asad Ullah Khan and others on 12 July 2023, a total of 75 influential large language models appeared from 2019 to 2023. Among them, they divided these large language models into 9 different application fields, namely General purpose, Medical, Education, Science, Maths, Law, Finance, Robotics, and Coding.

In addition, the model framework is divided into three categories: decoder-only language model, encoder-only masked language model and encoder-decoder model[Fu *et al.*, 2023]. The decoder-only language model can perform zero-shot program synthesis by generating in a left-to-right fashion. In the review statistics of this article, we found that the hybrid technology of large language model and fuzzing test uses more decoder-only language models, such as InCoder [Fried *et al.*, 2023], CodeX [Chen *et al.*, 2021]

On the other hand, CodeGen [Nijkamp *et al.*, 2023] is an encoder-decoder model. It was developed by Salesforce Research and is designed to handle natural language processing and program generation tasks. The CodeGen model combines an encoder and decoder architecture, which makes it particularly effective at understanding and generating text, including programming languages.

Encoder-decoder models are better able to handle complex language understanding and generation tasks than decoder-only or encoder-only models. CodeGen not only generates code, but also understands and processes more complex programming language structures and logic. This model is particularly suitable for tasks that require a deep understanding of code semantics and structure, such as code translation, code summarization, and complex code generation tasks.

2.2 Fuzzing Test

The first fuzzing tool was created by Miller et al. (1990) and was originally designed to test the software and system reliability [Chen *et al.*, 2018]. Ever since its introduction in the early 1990s, fuzzing has become a widely used technique to test software correctness and dependability [Manès *et al.*, 2021]. Before 2005, this technique is still in the early stage of development. Most of the fuzzing test in this period are black-box fuzzing using random mutation. From 2006 to 2010, some fuzzing systems adopted the taint analysis techniques, and the symbolic execution-based fuzzing was adequately developed. The number of fuzzing test systems were published, and some of them even could be used for commercial purposes. Between 2011 and 2015, the development of fuzzing test could be concluded with three main characteristics. First, in academic and industrial domains, coverage-guided fuzzing has become an important role. Second, different types of scheduling algorithms were used in fuzzing. Third, improving the efficiency of fuzzing by combining various of techniques became a trend. In 2016 and 2017, many fuzzing systems improved base on AFL(American fuzzy loop) [Zalewski, 2014], such as AFLFast[Böhme *et al.*, 2016] and AFLGo[Böhme *et al.*, 2017]. The trend of combining multiple techniques was still continuing. Moreover, the concept of machine learning was introduced to fuzzing[Chen *et al.*, 2018]. Until today, it is still developing and can be categorized into two major types, mutation-based fuzzing test and generation-based fuzzing test. Those two types of fuzzing test can be used for different kinds of software. According to the research about the art, science and engineering of fuzzing, current fuzzer, a program that performs fuzzing test on a program under test(PUT), can be separated into three classes, black-box fuzzer, white-box fuzzer, and grey-box fuzzer[Manès *et al.*, 2021]. The traditional black-box fuzzing test technique requires no prior information about the target PUT. It employs pre-defined mutation rules to mutate seed files and generate new erroneous inputs. In contrast to black-box fuzzing test, white-box fuzzing test necessitates the use of PUT information to guide the generation of test cases. Theoretically, white-box fuzzing test has the potential to cover every program path. Grey-box fuzzing test falls between black-box fuzzing test and white-box fuzzing test. It generates effective test cases with partial information of the PUT. A typical approach involves obtaining code coverage of the PUT during runtime and leveraging this information to adjust the mutation strategies[Liang *et al.*, 2018].

3 LLM Based Fuzzing test Analysis

The application scope of fuzzing test is wide, and it has been deployed in many software industry domains to test the reliability and security of software. Traditional fuzzing test has certain limitations, and to improve its efficiency and accuracy, some researchers in the field of technology have started to combine emerging large language models with fuzzing test. After a comprehensive analysis of LLMs-based fuzzers, as shown in Figure 1, we found that existing LLMs-based fuzzers usually introduce large language model technology into prompt engineering and seed mutation to enhance the performance of fuzzing test. In addition, we summarized the most commonly used metrics based on all technologies to evaluate the performance of LLMs-based fuzzers. As shown in Table 1, we classified models, benchmarks, and testing types based on all LLMs-based fuzzers, and conducted classification discussions and future work discussions on 14 literature in sections 4 and 5.

3.1 AI software

How do developers use large language models for fuzzing test?

In the fuzzing test of AI-type software, different software has different testing methods, and there are also different types of defense methods for different software attack methods. Lu Yan et al. proposed the technology ParaFuzz [Yan *et al.*, 2023], a testing framework for detecting contaminated samples in NLP models, which can enhance the backdoor defense of NLP models. They use the ChatGPT3.5 model for interpretation, convert trigger (and trigger characters in malware, etc.) removal into a prompt engineering task, and apply fuzzification technology to obtain the best interpretation prompt. In addition, Joshua Ackerman and George Cybenko used

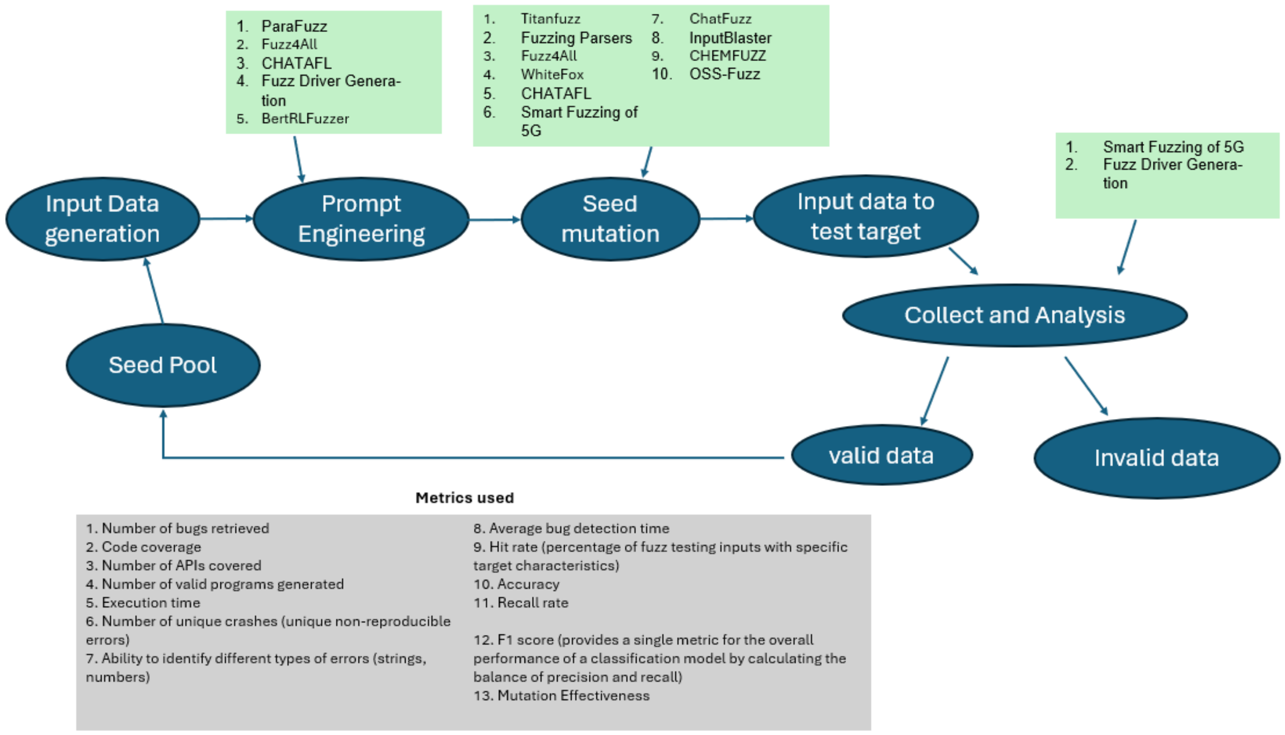
1. ParaFuzz
2. Fuzz4All
3. CHATAFL
4. Fuzz Driver Generation
5. BertRLFuzzer
1. Titanfuzz
2. Fuzzing Parsers
3. Fuzz4All
4. WhiteFox
5. CHATAFL
6. Smart Fuzzing of 5G
7. ChatFuzz
8. InputBlaster
9. CHEMFUZZ
10. OSS-Fuzz
1. Smart Fuzzing of 5G
2. Fuzz Driver Generation
Input Data generation
Prompt Engineering
Seed mutation
Input data to test target
Collect and Analysis
Seed Pool
valid data
Invalid data
Metrics used
1. Number of bugs retrieved
2. Code coverage
3. Number of APIs covered
4. Number of valid programs generated
5. Execution time
6. Number of unique crashes (unique non-reproducible errors)
7. Ability to identify different types of errors (strings, numbers)
8. Average bug detection time
9. Hit rate (percentage of fuzz testing inputs with specific target characteristics)
10. Accuracy
11. Recall rate
12. F1 score (provides a single metric for the overall performance of a classification model by calculating the balance of precision and recall)
13. Mutation Effectiveness

Figure 1: A General overview of LLMs-based Fuzzer

OpenAI's GPT-4 and OpenAI's text embedding model text-embedding-ada-002 to parse Natural language format specifications to automatically generate data instances that conform to these specifications [Ackerman and Cybenko, 2023]. This is to test whether different software can correctly handle the format of different types of data. ChatGPT3.5 and GPT-4 have powerful language generation and rewriting capabilities. Developers only need to handwrite the corresponding prompt template, and then ask the model to generate a large number of samples as input data based on these descriptions, which can achieve the effect of fuzzing test. Furthermore, Piyush Jha, et al proposed BertRLFuzzer [Jha *et al.*, 2023]. This is a fuzzer that combines multi-arm bandit agents, PPO agents and BERT models [Devlin *et al.*, 2019] to discover security vulnerabilities in web applications (such as SQL injection (SQLi), cross-site scripting (XSS), cross-site request forgery (CSRF)).They state that a large language model (Bert) can automatically learn syntactic fragments of attack vectors and use reinforcement learning to explore attack vector patterns of victim applications. This means that BertRLFuzzer automatically understands attack syntax fragments and then implements automated input testing through mutation operators given by reinforcement learning.

In addition, On 30 Dec 2022, Yinlin Deng and others developed TitanFuzz, which is the first technology to test deep learning libraries generated with a large language model. It achieves the effect of seed program mutation by using generative large language models and padded large language models, and then uses evolutionary algorithms for fuzzing test. Developers have developed a more powerful seed mutation process through the generative model CodeX and the infilling model CodeGen. We describe the details of TItanfuzz's seed mutation in the Prompt engineering and Seed Mutation section. With the rise of TitanFuzz, FuzzGPT was developed by Yinlin Deng and others on April 4, 2023. A core concept of FuzzGPT is that programs that historically trigger vulnerabilities may contain rare or valuable code components that are important when finding vulnerabilities. Therefore, FuzzGPT leverages CodeX and CodeGen models in large language models to automatically generate exception programs based on core concepts and is used to fuzz test deep learning libraries such as PyTorch[1] and TensorFlow[2]. Although TitanFuzz and FuzzGPT are both fuzzing test technologies based on CodeX and CodeGen models, they still have essential differences. Titanfuzz relies on prompt word engineering and strategies of seed generation and mutation to directly generate or rewrite code snippets. But FuzzGPT is not limited to the strategies used by Titanfuzz. It is a large language model fuzzer that uses methods such as few-shot learning [Brown *et al.*, 2020], zero-shot learning [Petroni *et al.*, 2019] and fine-tuning [Radford *et al.*, 2018]. It works by understanding historical error codes to generate more complex and specific edge-case code snippets.

Prompt engineering and Seed Mutation

Among traditional mutation fuzzers, the seed mutation technique is the most important because most model-free seed fuzzers generate a seed, which is an input to the PUT. The

[1]https://pytorch.org/

[2]https://www.tensorflow.org/?hl=zh-cn

purpose of modifying the seed is to generate a new test case that is mostly valid but also contains outliers to trigger the crash of PUT.

Traditional seed mutation technologies include Bit-Flipping[GPF, 2005], Arithmetic Mutation, Block-Based Mutation[Wang *et al.*, 2019], Dictionary-Based Mutation[Swiecki, 2010]. However, the seed mutation technology based on large language models has brought significant advantages to the performance of fuzzing test technology. It can understand and modify these data types through context awareness and natural language processing capabilities. It can maintain the validity of the file format while performing more complex mutation operations, such as rewriting code fragments, changing data structures, and even simulating complex user interaction.

Lu Yan et al. mentioned in the ParaFuzz technical literature that identifying synonyms and intersections of content words and function words is a specific challenge in the mutation process. Synonyms are needed to keep the meaning of the statement unchanged during the mutation process. Therefore, they gave a Meta prompt method to solve such problems by using ChatGPT to perform mutation operations. The specific operation is that the GPT model performs statement mutation operations based on a set of special prompts. In this way, data can retain its original meaning and be reorganized by other data. Furthermore, Titanfuzz leveraged CodeX and specific prompt words to generate large amounts of initial input data. Second, in order to obtain more diverse input data, they masked certain parts of the initial input data and then used fill-in models to fill in these masked parts. This enables the production of richer and more varied input data sets. TItanfuzz uses four methods to mask the content of the seed and then mutate the seed through CodeGen, namely parameter mutation operator, prefix/suffix mutation operator, and method mutation operator. According to these four methods, the program will mark the SPAN symbol at a specific position of the seed to prompt the model to randomly generate new content to achieve the effect of seed mutation.

3.2 Non-AI software

How do developers use large language models for fuzzing test?

In the field of fuzzing test for non-AI technology software, there are also similar techniques emerging, and even more. For example, Chunqiu Steven Xia, Matteo Paltenghi, and their team are dedicated to developing a universal fuzzing tester that can automate fuzzing test for projects written in different programming languages. Their proposed Fuzz4All [Xia *et al.*, 2024] is a fuzzing test technique that utilizes the large language model GPT4 and StarCoder [hug,] as input generation and mutation engines. Due to the extensive pre-training of the LLMs on various programming language examples, LLMs understand the syntax and semantics of different languages. Fuzz4All leverages this capability to conduct fuzzing test on projects, programs, compilers, and even API libraries written in different languages, greatly improving the universality of fuzzing test. Similarly, the white-box fuzzing test technique for compilers called WhiteFox [Yang *et al.*, 2023] published in October 2023, proposed by Yuanyang Chen, Jiawei Liu, and their team. It is also built upon two large language models, Gpt4 and StarCoder. WhiteFox divides these two models into an analytical LLM and a generative LLM. The former is used to analyze low-level optimized source code and specify requirements for high-level test programs that can trigger optimizations. The latter is responsible for generating test programs based on the requirements. Fuzz4All and WhiteFox utilize the same large language models and have consistent main testing targets, exhibiting several commonalities. GPT4 possesses powerful natural language understanding and analysis capabilities. However, the cost of input generation using such a large-scale model would be significantly high. Therefore, in WhiteFox and Fuzz4All, GPT4 is utilized as an analytical LLM technology for understanding, analyzing requirements, or extracting information. On the other hand, StarCoder is employed as a smaller model suitable for efficient continuous input generation. As a result, both techniques use StarCoder as a generative LLM.

ChatGpt[3] developed by OpenAI, along with a series of powerful engines such as Gpt3.5-turbo and Gpt4, are widely utilized in this field. For example, Cen Zhang and his team proposed a fuzzing test generator based on the LLMs Gpt3.5 and Gpt4 in august 6, 2023, which is used for fuzzing test of library APIs. This technology is more universal and lightweight. 64% of the issues can be completely automated, and if a manual semantic validator is added, this number will increase to 91% [Zhang *et al.*, 2023].

In addition, the fuzzing test technique for network protocols, ChatAFL [Meng *et al.*, 2024], utilizes LLM as an assistant and was developed by Ruijie Meng and his team. This technique incorporates Gpt-3.5 turbo. Specifically, the protocol fuzzing tool interacts systematically with LLM, enabling the fuzzing tool to present highly specific tasks to LLM. Some developers are dedicated to enhancing grey-box fuzzing test using large language models. The ChatFuzz [Hu *et al.*, 2023] technique developed by Jie Hu, Qian Zhang, and Heng Yin adopts a novel fuzzing test framework that combines generative AI with grey-box fuzzing test. In addition to the traditional grey-box fuzzing test process, a chat mutator selects a seed from a seed pool and prompts similar inputs from the ChatGpt model endpoint. Then the grey-box fuzz tester will evaluate those inputs, some inputs will be kept as new seeds for further exploration. The quality of seeds generated by the chat mutator is closely related to the model configuration of ChatGpt.

ChemFuzz [Qiu *et al.*, 2023] is a technique for fuzzing test software in the field of quantum chemistry. It was developed by Feng Qiu and his team and was published in October 2023. This technique utilizes the large language models GPT-3.5, Claude-2[Antropic, 2023], and Bart to provide valuable domain-specific chemical knowledge, enabling the generation and mutation of input files with grammar and semantic validity. ChemFuzz demonstrates significant improvements over its competitors when tested on specific quantum chemistry software, achieving a code coverage of 17.4% and identifying 40 unknown vulnerabilities. These results serve as

[3]https://openai.com/chatgpt

strong evidence of the success of this technology.

In the field of mobile applications, on October 23, 2024, Zhe Liu et al. proposed InputBlaster [Liu *et al.*, 2023], a technology based on generative large language models, Chatgpt and UIAutomator[4]. It utilizes LLM to automatically generate abnormal text inputs for testing mobile applications. This technology does not directly generate target inputs by LLM. Developers consider this approach to be inefficient and the interaction cost with LLM to be too high for their goals. Therefore, InputBlaster uses LLM to generate test generators (code snippets), where each generator can generate a batch of unconventional text inputs using the same mutation rule. The technology has achieved success in the field of mobile application testing. After testing 31 popular applications on Google Play, InputBlaster showed a significant improvement in bug detection rate, with a remarkable 136% increase compared to common and state-of-the-art testing methods. The bug detection rate of InputBlaster reached 78%.

As mentioned before, ChatGpt can understand tasks based on task examples or simple natural language instructions. In other words, it is an excellent zero-shot and few-shot learner. The learning capability of ChatGpt is a crucial reason why developers choose to utilize it. Additionally, as a generative LLM, it efficiently generates inputs based on prompts. In comparison to traditional fuzzing test, it exhibits improved performance. In subsequent sections, we will further discuss the performance of these techniques.

Large language models developed by Google are also applied to enhance fuzzing test techniques. For example, the proposal by Google's Open Source Security Team combines Google's developed large language model with the existing fuzzing test system OSS-Fuzz [Google, 2023], which has been running for many years, to test if it can enhance the performance of OSS-Fuzz. Based on the results obtained from the tests, it can be observed that OSS-Fuzz connected with LLM achieves higher code coverage for project detection and can handle more tasks.

In the field of 5G wireless software, in September 2023, Huan Wu, Brain Fang, and Fei Xie, published a technique that involved using the existing AFL++ fuzzing test system in combination with the large language model Google Bard to test OpenAirInterface5G, an open-source software framework for developing 5G wireless communication systems [Wu *et al.*, 2023]. It is worth mentioning that this technique utilizes the assistance of the large language model to automatically decipher and document the meanings of the parameters used for fuzzing test in the OAI5G codebase.

Prompt engineering and Seed Mutation

In the technique of fuzzing test for non-AI software using LLMs, prompt engineering and seed mutation remain crucial, but they differ from traditional fuzzing test approaches. Fuzz4All cleverly combines prompt engineering and seed mutation. It initially accepts arbitrary user inputs as initial prompts. As user prompts can be complex and lengthy, Fuzz4All first utilizes an LLM to extract concise yet informative prompts from the provided information. Each candidate prompt is then sent to a generative LLM to generate code snippets, also known as seeds, for fuzzing test. To avoid generating identical fuzz test inputs, Fuzz4All continually updates the input prompts in each iteration. It uses previously generated prompts as examples, demonstrating the types of inputs the user desires the model to generate.

The technique InputBlaster, which utilizes large language models to generate specialized text inputs for detecting mobile applications, incorporates contextual information from the user interface of the mobile app as prompts and generates code snippets as test generators. Each generator can produce a batch of unconventional text inputs based on the same mutation rule. Additionally, the LLM is requested to output its inferred constraint conditions, which are then used to generate mutation rules for the next iteration.

In emerging technologies, the integration of prompt engineering from LLMs with seed mutation from traditional fuzzing test is skillfully achieved.

3.3 Comparison between LLMs-based fuzzer and traditional fuzzer

Compared with traditional fuzzers, LLMs-based fuzzer have the following advantages:

Higher API&Code coverage:

By comparing TitanFuzz with the current state-of-the-art API-level (such as FreeFuzz [Wei *et al.*, 2022] and DeepREL [Deng *et al.*, 2022]) and model-level (such as LEMON [Wang *et al.*, 2020] and Muffin [Gu *et al.*, 2022]) fuzzers, it was found that TitanFuzz's API coverage in TensorFlow and PyTorch increased by 91.11% and 24.09% respectively. As model-level fuzzers, LEMON and Muffin use a small part of the hierarchical API (such as Conv2d), so their coverage is low. TitanFuzz can generate arbitrary code by combining generative (Codex) and infill (InCoder) large language models (LLMs) to achieve optimal API coverage.

CHATAFL, as a fuzzing test technique for network protocols based on large language models, exhibits higher average code coverage compared to state-of-the-art fuzz testers of the same kind, such as AFLNET [Pham *et al.*, 2020] and NSFuzz [Qin *et al.*, 2023], which do not utilize LLM technology. In comparison to AFLNET, CHATAFL achieves an average of 5.8% more branch coverage. When compared to NSFuzz, this number increases to 6.7%. This indicates that CHATAFL has a wider detection range and a higher chance of discovering unknown bugs.

Generate more efficient programs:

In terms of overall code, TitanFuzz achieved 20.98% and 39.97% code coverage on PyTorch and TensorFlow respectively, far exceeding DeepREL and Muffin. Compared with DeepREL, TitanFuzz's code coverage in PyTorch and TensorFlow increased by 50.84% and 30.38% respectively. Although TitanFuzz has a higher time cost, using only the seed generation function and testing against the APIs covered by DeepREL is significantly better than DeepREL and takes less time, which shows the advantage of directly using LLMs to generate high-quality seeds.

[4]https://developer.android.com/training/testing/other-components/ui-automator

Methods	Domain classification	Models	Benchmarking Dataset	Test type
TitanFuzz[5]	AI software	CodeGen	Pytorch, TensorFlow	Black-Box
FuzzGPT[6]	AI software	CodeX, CodeGen	Pytorch, TensorFlow	Black-Box
ParaFuzz	AI software	ChatGPT	Amazon Reviews, SST-2, IMDB, AGNews	Black-Box
Fuzzing Parsers	AI software	OpenAI's GPT-4	Tomita grammars, other novel grammars	Black-Box
BertRLFuzzer	AI software	BERT, Reinforcement Learning	victim websites, Other fuzzing test tools'benchmark	Grey-Box
Fuzz Driver Generation[7]	Non-AI software	GPT-3.5, GPT-4	NAIVE-K, BACTX-K	Black-Box
Fuzz4All[8]	Non-AI software	GPT-4, StarCoder	Six input languages and nine systems (SUTs)	Black-Box
WhiteFox[9]	Non-AI software	GPT-4, StarCoder	PyTorch Inductor, TensorFlow Lite, TensorFlow-XLA, LLVM	White-Box
OSS-Fuzz[10]	Non-AI software	Google LLM	tinyxml2, OpenSSL	Black-Box
CHATAFL[11]	Non-AI software	GPT-3.5 turbo	PROFUZZBENCH	Grey-Box
InputBlaster	Non-AI software	Chatgpt, UIAutomator	18 baselines from various aspects	Black-Box
Smart Fuzzing of 5G	Non-AI software	Google Bard, ChatGPT	OAI5G configuration file	Black-Box
CHATFUZZ[12]	Non-AI software	GPT-3.5 turbo	Unibench, Fuzzbench, LAVA-M	Grey-Box
CHEMFUZZ	Non-AI software	GPT-3.5, claude-2	Siesta(Quantum chemistry software)	Grey-Box

Table 1: Overview of core techniques in the important literature

Found more complex errors:

Traditional fuzzers usually randomly generate test cases based on given rules or methods. Such methods lack an in-depth understanding of the code structure, logic, and context. Therefore, they may not be effective in exploring complex programming patterns or identifying advanced vulnerabilities. Additionally, traditional fuzzers typically do not leverage historical bug data or programming patterns. In contrast, LLMs can learn from a large amount of historical code and bugs, thereby being able to simulate past bug patterns and discover new vulnerabilities. In FuzzGPT's experiments, a total of 76 bugs were detected, 61 of which were confirmed, including 49 confirmed as previously unknown bugs (6 of which have been fixed). More importantly, FuzzGPT detected 11 new high-priority bugs or security vulnerabilities. This shows that LLMs-based fuzzers can find deeper programming vulnerabilities.

For instance, when testing the same target using CHATAFL, NSFUZZ, and AFLNET, with the same number of runs and time, CHATAFL discovered 9 new vulnerabilities in the test target. In contrast, NSFUZZ only found 4 vulnerabilities, while AFLNET found three vulnerabilities.

Fuzz4All demonstrates superior performance. It has been able to discover 76 bugs in widely used systems such as GCC, Clang, OpenJDK, and 47 of these bugs have been confirmed as previously unknown vulnerabilities.

Increased automation:
Traditional fuzzing test requires a significant amount of time, effort, and manual labor. Seed generation is an indispensable part of fuzzing test, and creating diverse and effective inputs requires a lot of manual work and expertise. Additionally, using existing seeds to generate new inputs through mutation also consumes time. Automating fuzzing test using large language models can address these issues. InputBlaster leverages the automation capabilities of LLMs to generate high-quality seeds based on input prompts. After each test, it mutates the seeds based on different prompts, enabling the generation of new seeds for further testing. Implementing automated fuzzing test can save considerable costs, and this is expected to be a major trend in the future.

4 Discussion on Future work

4.1 Two types of LLMs-based fuzzers

LLMs-based fuzzers can be divided into two major sections. The first part is to find the historical data set of the object being tested, classify the historical error code fragments and vulnerabilities and provide them to the model for learning, and finally train a professional fuzzing test model to test the object. Another section is to introduce LLMs into certain steps in the traditional fuzzing process to improve the performance of the fuzzer. For example, TitanFuzz mentioned using CodeX and CodeGen to change the seed mutation process in traditional fuzzing test to improve code coverage and test success rate. On the other hand, FuzzGPT learned historical dataset that causes bugs and vulnerabilities to the deep learning system. However, the performance comparison between FuzzGPT and Titanfuzz shows that FuzzGPT is superior to TItanfuzz in terms of code coverage and efficiency. Therefore, this means that in the future development of the LLMs-based fuzzer field, it is more promising to let the model learn historical data and become a professional fuzzer, rather than letting the model change the operation process of the traditional fuzzer.

4.2 Discussion of pre-training data

In the aforementioned context, training specialized models for fuzzing test represents a prospective research direction. However, challenges about the sufficiency and quality of pre-training data are notable. Yin Lin and colleagues have indicated that biases in datasets can arise, potentially reflecting societal inequalities or the prejudiced perspectives of data scientists[Lin *et al.*, 2020]. This suggests that the quality of pre-training datasets used for models might be skewed. Jean Kaddour and others have pointed out that the repetitiveness and scale of pre-training datasets can impact the performance of large language models (LLMs), such as the presence of semantically near-duplicate content which could increase the model's dependency on the data [Kaddour *et al.*, 2023]. LLM-based fuzzers will likely encounter these issues in the future. When the volume of data is inadequate or of poor quality, LLMs may not achieve exceptional performance in specific domains. Whether LLM-based fuzzers can surpass traditional fuzzers in terms of performance remains to be observed. Addressing these challenges will be essential for the future success of LLM-based fuzzers.

4.3 Discussion of time consuming

LLMs are highly complex deep learning models that require a large amount of computing resources to generate code. This complex calculation process usually takes more time than traditional fuzzing test methods. For example, TitanFuzz not only generates code but also generates high-quality, meaningful program snippets, which may involve multiple iterations and adjustments to ensure that the generated program meets specific quality standards. Additionally, to generate unique and diverse programs, additional checks and filters are required to ensure that existing code snippets are not duplicated. Furthermore, in the context of deep learning libraries, API calls may involve complex data structures and algorithms. Generating code that effectively calls these APIs and triggers potential errors may require more complex logic and longer processing time.

4.4 Discussion of Evaluation Framework for Large Language Model-based fuzzing test

The criteria for evaluation are mainly customized for different testing objectives. The ultimate measure of a fuzzer is the number of distinct vulnerabilities it discovers. In addition, it is very common to assess the performance of a fuzzer based on code coverage [Klees *et al.*, 2018]. Many techniques measure code coverage, and some techniques even use code coverage as the only evaluation criterion, such as FairFuzz [Lemieux and Sen, 2018].Fuzz4All mentioned that code coverage measurement was used to evaluate the performance of the fuzzer. To maintain consistency, line coverage for each evaluation target was reported. However, there is no fundamental reason to directly associate maximizing code coverage with discovering vulnerabilities. In the study published by Shiyi Wei et al., they mentioned that they conducted a statistical analysis of 32 fuzzy testing papers, examining their experimental methods. They concluded that there is an urgent need for well-designed and thoroughly evaluated benchmark suites for fuzzing test [Klees *et al.*, 2018]. Many traditional fuzzing test evaluation criteria have been used to assess LLMs-based fuzzer. Therefore, we need a universal and more detailed evaluation framework specifically designed for LLMs-based fuzzer. The new evaluation framework should include, but not be limited to, evaluations of different LLMs used by fuzzers. It should compare the efficiency of fuzzers utilizing LLM with traditional fuzzers under consistent parameters, environments, and test subjects. This evaluation framework should be universal and applicable to the majority of LLMs-based fuzzer. It will be a crucial standard for measuring the performance of such technologies in the future.

4.5 Discussion of achieving full automation

LLMs-based fuzzers significantly reduce manual labor compared to traditional fuzzers. ChatFuzz leverages prompt engineering from large language models for seed selection and mutation. Fuzz4All updates the LLM prompts after each iteration to avoid generating the same test inputs and to generate higher-quality inputs. Similar work in traditional fuzzing test requires developer intervention. In a Google Open Source blog, it was mentioned that manually copying results after testing a project with the traditional fuzzer OSS-Fuzz consumes a considerable amount of time, which was notably improved by integrating LLM with OSS-Fuzz. These techniques utilize the characteristics of large language models to some extent, making the developer's workload lighter. Currently, LLM-based fuzzers cannot fully achieve the complete automation of generating fuzzing test, but this is undoubtedly the direction of future development.

5 Conclusion

This article provides an in-depth review and summary of fuzzing test technology based on large language models and covers a wide range of applications in AI and non-AI software fields. We summarized the frameworks and principles of different LLMs-based fuzzers, and described how these technologies introduce LLMs to enhance traditional fuzzing test technologies.

Compared with traditional fuzzers, the LLMs-based fuzzers provide superior API and code coverage, find more complex bugs, and improve the automation of fuzzing tests. Therefore, LLMs-based fuzzing technology has great potential in advancing the field of software testing. The insights and results of this survey are expected to be a valuable resource for researchers and practitioners in the field, guiding the development of more efficient, reliable, and automated fuzzing test methods by using the LLMs.

References

[Ackerman and Cybenko, 2023] Joshua Ackerman and George Cybenko. Large language models for fuzzing parsers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*, FUZZING 2023, page 31–38, New York, NY, USA, 2023. Association for Computing Machinery.

[Antropic, 2023] Antropic. Claude 2 — anthropic.com. https://www.anthropic.com/news/claude-2, 2023. [Accessed 30-01-2024].

[Böhme *et al.*, 2016] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[Böhme *et al.*, 2017] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[Brown *et al.*, 2020] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[Chen *et al.*, 2018] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.

[Chen *et al.*, 2021] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[Deng *et al.*, 2022] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational api inference, 2022.

[Deng *et al.*, 2023a] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

[Deng *et al.*, 2023b] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014*, 2023.

[Devlin *et al.*, 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[Fried *et al.*, 2023] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis, 2023.

[Fu *et al.*, 2023] Zihao Fu, Wai Lam, Qian Yu, Anthony Man-Cho So, Shengding Hu, Zhiyuan Liu, and Nigel Collier. Decoder-only or encoder-decoder? interpreting language model as a regularized encoder-decoder, 2023.

[Google, 2023] Google. AI-Powered Fuzzing: Breaking the Bug Hunting Barrier — security.googleblog.com. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html, 2023. [Accessed 25-01-2024].

[GPF, 2005] GPF. Welcome — vdalabs.com. https://www.vdalabs.com/, 2005. [Accessed 31-01-2024].

[Gu *et al.*, 2022] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing, 2022.

[Hu *et al.*, 2023] Jie Hu, Qian Zhang, and Heng Yin. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782*, 2023.

[hug,] StarCoder: A State-of-the-Art LLM for Code — huggingface.co. https://huggingface.co/blog/starcoder. [Accessed 25-01-2024].

[Jha *et al.*, 2023] Piyush Jha, Joseph Scott, Jaya Sriram Ganeshna, Mudit Singh, and Vijay Ganesh. Bertrlfuzzer: A bert and reinforcement learning based fuzzer. *arXiv preprint arXiv:2305.12534*, 2023.

[Kaddour *et al.*, 2023] Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models, 2023.

[Klees *et al.*, 2018] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing, 2018.

[Lemieux and Sen, 2018] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 475–485, 2018.

[Liang *et al.*, 2018] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[Lin *et al.*, 2020] Yin Lin, Yifan Guan, Abolfazl Asudeh, and HV Jagadish. Identifying insufficient data coverage in databases with multiple relations. *Proceedings of the VLDB Endowment*, 13(11), 2020.

[Liu *et al.*, 2023] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. Testing the limits: Unusual text inputs generation for mobile app crash detection with large language model. *arXiv preprint arXiv:2310.15657*, 2023.

[Manès *et al.*, 2021] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021.

[Meng *et al.*, 2024] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[Naveed *et al.*, 2023] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2023.

[Nijkamp *et al.*, 2023] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[Petroni *et al.*, 2019] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H. Miller, and Sebastian Riedel. Language models as knowledge bases?, 2019.

[Pham *et al.*, 2020] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[Qin *et al.*, 2023] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology*, 2023.

[Qiu *et al.*, 2023] Feng Qiu, Pu Ji, Baojian Hua, and Yang Wang. Chemfuzz: Large language models-assisted fuzzing for quantum chemistry software bug detection. In *23rd IEEE International Conference on Software Quality, Reliability, and Security. Accompany. (QRS 2023).*, 2023.

[Radford *et al.*, 2018] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training, 2018.

[Swiecki, 2010] Robert Swiecki. GitHub - google/honggfuzz: Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based) — github.com. https://github.com/google/honggfuzz, 2010. [Accessed 31-01-2024].

[Wang *et al.*, 2019] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[Wang *et al.*, 2020] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 788–799, 2020.

[Wei *et al.*, 2022] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source, 2022.

[Wu *et al.*, 2023] Huan Wu, Brian Fang, and Fei Xie. Smart fuzzing of 5g wireless software implementation. *arXiv preprint arXiv:2309.12994*, 2023.

[Xia *et al.*, 2024] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models, 2024.

[Yan *et al.*, 2023] Lu Yan, Zhuo Zhang, Guanhong Tao, Kaiyuan Zhang, Xuan Chen, Guangyu Shen, and Xiangyu Zhang. Parafuzz: An interpretability-driven technique for detecting poisoned samples in nlp. *arXiv preprint arXiv:2308.02122*, 2023.

[Yang *et al.*, 2023] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. White-box compiler fuzzing empowered by large language models, 2023.

[Zalewski, 2014] Michał Zalewski. american fuzzy lop — lcamtuf.coredump.cx. https://lcamtuf.coredump.cx/afl/, 2014. [Accessed 25-01-2024].

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