Short Paper: Systematic Bug Reproduction with Large Language Model

Sanghyun Park, Haeun Lee, and Sang Kil Cha(✉)

KAIST, Daejeon, Republic of Korea
{sanghyun.park,haeun.lee,sangkilc}@kaist.ac.kr

Abstract. Analyzing 1-day vulnerabilities is a critical task in software security, but it is often challenging to reproduce the bugs due to the lack of information about the vulnerabilities. In this paper, we explore how Large Language Models (LLMs) can be leveraged to generate inputs that trigger specific vulnerabilities. There are two main challenges: LLMs must (1) correctly analyze the target vulnerability and (2) identify relevant fields to generate meaningful program inputs. We address these challenges through a three-stage prompting approach, where we provide necessary information at each stage, guiding the LLM to ultimately generate input for reproducing the target bug. By using these generated inputs as seeds for directed fuzzing, we show that our strategy can effectively generate useful inputs for vulnerability reproduction.

Keywords: bug reproduction · large language models · directed fuzzing

1 Introduction

Analyzing known (1-day) vulnerabilities is challenging because finding the program inputs that trigger the vulnerabilities requires considerable effort and time [17]. Specifically, this process demands a deep understanding of the program and its input formats, along with the expertise to manipulate inputs to satisfy complex constraints.

Despite numerous studies on automated 1-day vulnerability analysis, efficiency limitations still persist. Directed fuzzing [2,9,10] is a promising technique for reproducing known vulnerabilities by gradually mutating inputs to reach the target location. However, recent studies [11,12] show that directed fuzzers struggle to quickly generate inputs for vulnerabilities that do not align with the heuristics they employ. Dynamic symbolic execution [4,5] and its variant, directed symbolic execution [14], are effective for reproducing bugs, but they do not scale well for large programs due to the path explosion problem.

To address these challenges, we propose a novel technique based on Large Language Models (LLMs) for efficient and automatic 1-day vulnerability analysis. As LLMs generate outputs based on learned data, they do not rely on specific heuristics. Notably, they are capable of performing various tasks that

J. Garcia-Alfaro et al. (Eds.): ESORICS 2024 Workshops, LNCS 15264, pp. 39–48, 2025.
https://doi.org/10.1007/978-3-031-82362-6_3

previously required human expertise in a short amount of time due to their extensive training on large datasets [3,6].

There are two main challenges when using LLMs for vulnerability analysis. First, the LLM may fail to understand the target vulnerability and its root cause. Second, it may struggle to identify the input fields relevant to the vulnerability.

In this paper, we address these challenges using a systematic three-stage prompting method. The key intuition of our approach is to guide the LLM through a series of prompts that build on each other to generate inputs that can be used in further analysis, for example, as initial seeds in directed fuzzing. To begin, we provide the LLM with information regarding the target vulnerability and ask the LLM for its root cause. Next, we ask the LLM to identify the fields related to the vulnerability. Finally, we provide a small program input and instruct the LLM to generate a bug-triggering input by modifying or inserting the fields identified in the previous stage. In this way, we break down the complex problem of finding a bug-triggering input into smaller, manageable tasks, allowing the LLM to progress in a step-wise fashion by combining the information from the prompts with the answers from previous stages. Although the generated inputs may not be the exact bug-triggering inputs due to the limitations of the LLMs, they are expected to serve as effective initial seeds for directed fuzzing.

We implement these ideas in a framework, named LLM1dFuzz, and evaluate its effectiveness by applying it to 15 real-world programs with known vulnerabilities. The LLM successfully analyzes the causes of vulnerabilities and identifies the related fields in 8 (53%) and 7 (47%) programs, respectively. Moreover, using the LLM-modified inputs as initial seeds for directed fuzzing improved the fuzzing results in 37.5% of the programs while showing the same performance in 36.7% and worse in 25.8%. Our contributions are:

- We propose an LLM-based bug reproduction strategy, named LLM1dFuzz.
- We address the challenges of applying LLMs in bug-triggering input generation through a systematic three-stage prompting procedure.
- We publicize our code, prompts, and experimental results in support of open science: https://github.com/SoftSec-KAIST/LLM1dFuzz.

2 Related Work

1-day Vulnerability Reproduction. There have been several studies on reproducing 1-day vulnerabilities from patched binaries. 1dVul [20] combines directed fuzzing and directed symbolic execution to reach a target location in the binary effectively. It heuristically identifies target locations by analyzing the binary diff between the vulnerable and patched binaries. 1dFuzz [22] improves upon 1dVul by locating target locations using a well-known code pattern of security patches, named Trailing Call Sequence. Both approaches use patched binaries to reproduce 1-day vulnerabilities, while our study leverages source-level patches to direct the LLM to generate inputs that reproduce known vulnerabilities. Therefore, ours is orthogonal to these binary-level approaches and can be

complementary to them. Although not directly related to reproducing 1-day vulnerabilities, there are studies that use directed fuzzing to reproduce bugs from patches [23,24], which are complementary to our approach.

Input Generation with LLM. Recent research has been actively exploring the use of LLMs to generate or modify program inputs for fuzzing. However, they mainly focus on the LLMs' ability to generate well-formed inputs to find arbitrary bugs, while our study focuses on generating inputs specifically tailored to known vulnerabilities. Asmita *et al.* [1] demonstrated the effectiveness of using an LLM to generate diverse initial seeds. However, they primarily focused on generating initial inputs that conform to the input format of the program by simply providing the name of the program to the LLM. ChatAFL [15] leverages an LLM to perform stateful fuzzing, but it does not utilize the information provided in the prompts in subsequent prompts. Fuzz4All [21], similarly to our technique, includes official documentation and related code in their prompts, but it cannot be directly applied to reproducing known vulnerabilities.

3 Methodology

This section outlines the core methodology of our research, including a three-stage prompting process and its specific design points. Figure 1 shows the templates of our three-stage prompts. Stage 1 checks if the LLM can identify the root cause of the bug. Stage 2 determines if the LLM can specify the fields related to invoking the bug. Stage 3 assesses whether the LLM can modify a minimal input to create an input that triggers the bug. Note that each subsequent stage's prompt uses the information from the previous stages and their responses. In our methodology, if the LLM fails to answer the question, we provide the correct answer to continue with the next stage.

Stage 1: Vulnerability Analysis The Stage 1 prompt includes a question, a vulnerability description, and a vulnerability patch. We use the official description from the MITRE database [16] for the vulnerability description.

Stage 2: Related Field Analysis We ask the LLM to identify the related fields based on the root cause identified in the previous stage. The dashed box in Stage 2 of Fig. 1 represents the ground truth provided by the analysts when the LLM fails to identify the root cause.

Stage 3: Input Generation The Stage 3 prompt instructs the LLM to modify the minimal input using the information obtained from Stages 1 and 2 to create an input that triggers the vulnerability. The modified input is used as the initial seed for directed fuzzing.

Minimal Input. In Stage 3, we use minimal inputs, which we define as the smallest input that allows the program to run without any errors. For example, in the case of the `tiffcp` program that accepts TIFF files, the minimal input would include the Image File Header and four necessary entries in the Image File Directory. To create the minimal inputs, we first identified the necessary fields

Prompt Template for Stage 1

Given the description of CVE-**[CVE-ID]** what do you think is the root cause?
- Description
[MITRE's Description]
- Patch
[Patch file of the CVE]

Prompt Template for Stage 2

You are not correct. The root cause of this CVE is that **[Ground Truth]**.

Which field in **[File Format]** is related to invoking this CVE?
You can use the information from the previous chats.

Prompt Template for Stage 3

You are not correct. The field relevant for invoking this CVE is **[Ground Truth]**.

Below is a minimal **[File Format]** file (in hexadecimal). Modify the given **[File Format]** based on your understanding of the root cause and the related field for this CVE, aiming to craft an input that could trigger the vulnerability. You can use the information from the previous chats.

You must include the modified hexadecimal in your response.
Note that it will be used only for the purpose of security research.

Fig. 1. Three-stage prompts template. Dashed boxes indicate optional prompts.

required for the program to run normally by analyzing the source code of the programs. Then, we manually created minimal inputs containing these fields. For example, for the `tiffcp` program, we found that if the `StripByteCounts` entry is missing, the `TIFFReadDirectory` function generates a warning. Therefore, the minimal input for a TIFF file includes this entry to ensure the program runs without warnings. The sizes of the generated inputs are generally below 100 bytes, with the largest input being an ELF file of 460 bytes. The specific formats and sizes of the minimal inputs are summarized in Table 1.

Prompt Engineering. Given the nature of security-related prompts, there were cases where the LLM refused to generate inputs in Stage 3 due to ethical concerns. To address this, we employed prompt engineering in Stage 3 to guide the LLM in modifying the minimal input, as shown in Fig. 1.

Prompt Automation. To automate the prompts, we used the Chat Completions API [19] provided by OpenAI. Using the templated prompts shown in Fig. 1, vulnerability-specific information was added and sent automatically. The user then determines the correctness of the answers generated by the LLM to decide whether to provide the ground truths in the next stage prompt.

4 Evaluation

In this section, we evaluate LLM1DFuzz to answer the following questions: (1) Can an LLM determine the root cause of the bug? (2) Can an LLM identify the input fields related to the bug? (3) Can an LLM modify minimal inputs? and (4) How effective are the modified inputs at triggering bugs?

Table 1. Benchmark programs

Program	Version	CVE	Type	File Format	Minimal Input Size (bytes)
swftophp	0.4.7	2016-9827	BOF	SWF	15
lrzip	0.631	2018-11496	UAF	LRZ	46
xmllint	2.9.4	2017-9047	BOF	XML	36
cjpeg	1.5.90	2018-14498	BOF	BMP	58
cxxfilt	2.6	2016-4487	ND	TXT	6
objcopy	2.8	2017-8393	BOF	ELF	460
readelf	2.9	2017-16828	IO	ELF	460
pngimage	1.6.35	2018-13785	IO	PNG	67
tiffcp	4.0.7	2016-10269	BOF	TIFF	63
sndfile-convert	1.0.28	2018-19758	BOF	WAV	45
openssl	1.1.0c	2017-3735	BOF	DER	201
lua	5.4.0	2020-24370	IO	LUA	2
php	7.3.6	2019-11041	BOF	JPEG	243
sqlite3	3.30.1	2019-19923	ND	SQL	9
pdfimages	0.73.0	2019-7310	BOF	PDF	137

4.1 Evaluation Setup

We selected 15 open-source programs that take 14 different file formats as input as the benchmark for our evaluation. The programs were chosen from a commonly used fuzzing benchmark [8] and benchmarks from previous directed fuzzing papers [2,9,10]. We randomly selected one CVE for each program, resulting in a total of 15 vulnerabilities for the experiment, as summarized in Table 1. We used GPT-4 Turbo [18] as the LLM. To mitigate the randomness of the LLM, we repeated the experiment 10 times per program. The success of Stages 1 and 2 was determined by comparing the responses generated by the LLM with the detailed answers about the vulnerabilities analyzed by our research team.

Considering that the LLM often does not modify the minimal input in Stage 3 for ethical reasons, the experiment was repeated up to 100 times until 10 modified inputs were generated. However, despite prompt engineering, the LLM only returned partially modified inputs for the `objcopy` and `readelf` programs, which take large and complex ELF structures as input, even after 100 prompts in Stage 3. For these programs, we manually applied the partial modifications to the minimal inputs, completing 7 and 10 initial seeds for fuzzing, respectively.

To evaluate the generated inputs, we used SelectFuzz (commit 6da35e0d) [13] as a directed fuzzer and AFL++ (v4.07c) [7] as an undirected fuzzer. For each program, 10 modified inputs were fuzzed five times for 24 h each. We labeled the inputs based on code coverage, identifying the highest and lowest, with the rest randomized (e.g., random-1, random-2, ..., random-8). We ran the experiments on a server machine with 88 Intel Xeon E5-2699 v4 (2.2GHz) CPU cores and 512GB of memory. Each fuzzing session was run in an isolated Docker container with one core and 4GB of memory assigned. A total of 40 fuzzing sessions were run simultaneously, utilizing 40 out of the 88 logical cores.

Table 2. Result of each stages

	swftophp	lrzip	xmllint	cjpeg	cxxfilt	objcopy	readelf	pngimage	tiffcp	sndfile-convert	openssl	lua	php	sqlite3	pdfimages
Stage 1 (# of Successes)	3	0	10	6	0	10	0	10	0	0	10	10	10	10	0
Stage 2 (# of Successes)	10	0	9	0	0	9	0	6	0	0	10	10	0	10	0
Stage 3 (# of Prompts)	43	10	12	10	13	100	100	17	24	21	100	15	16	10	10

4.2 Root Cause Analysis

Table 2 shows the evaluation results for each stage. Out of 15 programs, the LLM successfully identified the cause of the vulnerability in more than half of the attempts for 8 programs (53%). This shows the LLM's ability to infer the root cause of vulnerabilities based on the provided information and patch details. Compared to manual analysis by humans, the high success rate of the LLM suggests a high potential for vulnerability analysis.

The results showed that the majority of the LLM's answers heavily relied on the vulnerability descriptions and patch information provided in the prompts. For instance, in the case of CVE-2018-11496, the LLM incorrectly accepted erroneous vulnerability information listed by MITRE and returned unrelated information as the root cause of the vulnerability. Similarly, for CVE-2016-4487, multiple vulnerabilities were patched simultaneously in one commit, leading to irrelevant information being included in the patch data provided to the LLM. As a result, the LLM struggled to cherry-pick relevant information pertinent to CVE-2016-4487 and failed to pinpoint the root cause. This highlights the importance of providing accurate vulnerability information and patches to enhance the inference capabilities of the LLM.

4.3 Related Fields Identification

In Stage 2, the LLM successfully identified the fields related to invoking the vulnerability in more than half of the attempts for 7 out of 15 programs (46.7%). It is remarkable that the LLM could identify related fields based on the small amount of information provided in the prompts. This shows the potential for higher success rates with more comprehensive training data in the future.

By analyzing the failure cases, we found that the LLM often struggled to match variables in the patches to the corresponding input format fields. For example, in the case of CVE-2019-11041, while the LLM recognized that the `Thumbnail->size` variable in the patch was relevant to the vulnerability, it could not match which field in the JPEG format corresponded to this variable. Additionally, when multiple fields were related to invoking a vulnerability, the LLM frequently failed to identify all the relevant fields. For CVE-2016-10269, which required three fields to trigger the vulnerability, the LLM only identified one of them. This shows that the LLM has difficulties in matching variables in the code to input format fields and identifying multiple related fields.

Table 3. Fuzzing results of AFL++

		LLM-generated Inputs									
	Minimal Input	Best Cov.	Worst Cov.	Rand. 1	Rand. 2	Rand. 3	Rand. 4	Rand. 5	Rand. 6	Rand. 7	Rand. 8
swftophp	13(5)	14(5)	50(5)	15(5)	7(5)	21(5)	6(5)	13(5)	8(5)	8(5)	52(5)
lrzip	18(5)	4527(5)	2569(5)	1964(5)	4799(5)	4513(5)	2893(5)	1752(5)	5448(5)	1189(5)	1468(5)
xmllint	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	65393(4)	N.A.(1)	76979(3)	N.A.(0)	N.A.(0)	77274(3)	N.A.(0)
cjpeg	80769(3)	5487(4)	N.A.(1)	25(5)	34405(4)	7175(5)	191(5)	30417(3)	11927(4)	31348(3)	N.A.(2)
cxxfilt	1108(5)	581(5)	508(5)	172(5)	662(5)	895(5)	1366(5)	1633(5)	684(5)	1045(5)	817(5)
objcopy	1005(5)	1233(5)	N.A.(1)	N.A.(2)	N.A.(2)	N.A.(0)	N.A.(2)	N.A.(2)	-	-	-
readelf	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
pngimage	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
tiffcp	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
sndfile-convert	N.A.(0)	227(5)	76(5)	674(5)	192(5)	12(5)	426(5)	7(5)	20(5)	537(5)	490(5)
openssl	N.A.(0)	N.A.(0)	-	-	-	-	-	-	-	-	-
lua	N.A.(0)	13558(5)	N.A.(0)	44(5)	174(5)	0(5)	N.A.(0)	69(5)	17267(4)	3385(5)	51(5)
php	2839(5)	9873(5)	N.A.(2)	N.A.(1)	30820(3)	13772(4)	6318(4)	N.A.(2)	34252(4)	N.A.(2)	59757(4)
sqlite3	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
pdfimages	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)

4.4 Modification of Minimal Inputs

In Stage 3, the LLM successfully generated 10 modified inputs for 12 out of 15 programs (80%). However, despite prompt engineering efforts (§3), there were cases where the LLM refused to modify the minimal inputs due to ethical concerns. For instance, for CVE-2017-3735, which takes DER files as its input format, only one modified input was generated after 100 prompt attempts. The LLM also failed to generate ELF files, which have complex structures, even after 100 prompts. Conversely, vulnerabilities with relatively simple input formats yielded 10 modified inputs within about 30 prompts. This indicates that the model has difficulty modifying the minimal inputs for programs with complex and lengthy input formats.

4.5 Effectiveness of Generated Inputs

Table 3 and Table 4 reveal the fuzzing results using inputs generated by the LLM as initial seeds. The numbers represent the median time taken to find the vulnerability (i.e., Time-to-Exposure (TTE)), where "N.A." denotes that the vulnerability was not found within the 24-hour time limit in more than half of the repetitions. The parentheses denote the number of successful repetitions out of 5, where the results marked with a dash (-) indicate that the fuzzer failed to perform static analysis, which made fuzzing infeasible.

There was a promising result when fuzzing `lua` with random-3 input as the initial seed. The vulnerability was triggered immediately, indicating that the LLM succeeded in generating the text-formatted input using the detailed vulnerability descriptions and patches.

When fuzzing with the LLM-generated inputs as initial seeds, AFL++ and SelectFuzz succeeded in reproducing the vulnerability faster for 33% (5/15) and 42% (5/12) of the programs, respectively, compared to using minimal input

Table 4. Fuzzing results of SelectFuzz

		LLM-generated Inputs									
	Minimal Input	Best Cov.	Worst Cov.	Rand. 1	Rand. 2	Rand. 3	Rand. 4	Rand. 5	Rand. 6	Rand. 7	Rand. 8
swftophp	45(5)	13(5)	983(5)	50(5)	27(5)	20(5)	21(5)	41(5)	18(5)	17(5)	690(5)
lrzip	15(5)	7065(5)	20796(5)	2043(5)	3327(5)	5923(5)	803(5)	1805(5)	11581(5)	5870(5)	1751(5)
xmllint	N.A.(0)	N.A.(1)	N.A.(0)	N.A.(0)	N.A.(1)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
cjpeg	32520(3)	100(5)	N.A.(0)	37(5)	187(5)	704(5)	42(5)	276(5)	220(5)	N.A.(0)	N.A.(0)
cxxfilt	695(5)	649(5)	1242(5)	498(5)	1048(5)	514(5)	965(5)	1216(5)	1077(5)	753(5)	1163(5)
objcopy	1392(5)	3006(5)	N.A.(1)	N.A.(1)	N.A.(0)	N.A.(0)	N.A.(2)	N.A.(0)	-	-	-
readelf	-	-	-	-	-	-	-	-	-	-	-
pngimage	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
tiffcp	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
sndfile-convert	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	354(5)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
openssl	-	-	-	-	-	-	-	-	-	-	-
lua	N.A.(0)	1099(5)	N.A.(0)	N.A.(1)	N.A.(1)	0(5)	N.A.(0)	N.A.(0)	853(5)	2228(5)	1616(5)
php	-	-	-	-	-	-	-	-	-	-	-
sqlite3	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)
pdfimages	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)	N.A.(0)

values. Except for the `lrzip` case, the inputs generated by the LLM improved fuzzing performance for programs with small minimal inputs, such as BMP and WAV. However, it was found to be less effective for modifying complex input formats like ELF and DER. This is because, despite knowing the fields related to invoking the vulnerability, it was challenging to determine which part of the minimal input represented in hexadecimal corresponded to those fields. In the cases of CVE-2017-16828 and CVE-2017-3735, the LLM modified arbitrary fields in the input file for illustrative purposes, ignoring the file format complexity.

For `lrzip`, it is suspected that the lack of information on the input format likely hindered the model's learning process. In fact, we found the LLM modifying fields unrelated to the vulnerability in `lrzip`. Additionally, it was observed that high code coverage of inputs does not necessarily enhance fuzzing performance. There were several inputs with lower code coverage that showed better fuzzing performance than those with high code coverage.

5 Conclusion

This paper discusses how LLMs can assist in generating inputs that trigger specific vulnerabilities. Although we did not achieve successful results for all programs, it is significant to find that LLMs can be utilized in 1-day vulnerability analysis and aid in generating inputs that trigger vulnerabilities.

Acknowledgements. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2021-II210177, 2021-0-00177, High Assurance of Smart Contract for Secure Software Development Life Cycle).

References

1. Asmita, Scott, Y.O.M., Tsang, R., Fang, C., Homayoun, H.: Fuzzing BusyBox: Leveraging llm and crash reuse for embedded bug unearthing. In: Proceedings of the USENIX Security Symposium (2024)
2. Böhme, M., Pham, V.T., Nguyen, M.D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 2329–2344 (2017)
3. Brown, T.B., et al.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems, pp. 1877–1901 (2020)
4. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the USENIX Symposium on Operating System Design and Implementation, pp. 209–224 (2008)
5. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 380–394 (2012)
6. Chen, M., et al.: Evaluating large language models trained on code. In: arXiv preprint arXiv:2107.03374 (2021)
7. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++ : Combining incremental steps of fuzzing research. In: Proceedings of the USENIX Workshop on Offensive Technologies (2020)
8. Hazimeh, A., Herrera, A., Payer, M.: Magma: a ground-truth fuzzing benchmark. Proc. ACM Measure. Anal. Comput. Syst. **4**(3), 1–29 (2020)
9. Huang, H., Guo, Y., Shi, Q., Yao, P., Wu, R., Zhang, C.: Beacon: Directed grey-box fuzzing with provable path pruning. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 36–50 (2022)
10. Kim, T.E., Choi, J., Heo, K., Cha, S.K.: DAFL: Directed grey-box fuzzing guided by data dependency. In: Proceedings of the USENIX Security Symposium, pp. 4931–4948 (2023)
11. Kim, T.E., Choi, J., Im, S., Heo, K., Cha, S.K.: Evaluating directed fuzzers: Are we heading in the right direction? In: Proceedings of the International Symposium on Foundations of Software Engineering (2024)
12. Lee, H., Yang, H.D., Ji, S.G., Cha, S.K.: On the effectiveness of synthetic benchmarks for evaluating directed grey-box fuzzers. In: Proceedings of the Asia-Pacific Software Engineering Conference, pp. 11–20 (2023)
13. Luo, C., Meng, W., Li, P.: SelectFuzz: Efficient directed fuzzing with selective path exploration. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 2693–2707 (2023)
14. Ma, K.K., Khoo, Y.P., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Proceedings of the International Static Analysis Symposium, pp. 95–111 (2011)
15. Meng, R., Mirchev, M., Böhme, M., Roychoudhury, A.: Large language model guided protocol fuzzing. In: Proceedings of the Network and Distributed System Security Symposium (2024)
16. MITRE Corporation: CVE-MITRE. https://cve.mitre.org
17. Mu, D., et al.: Understanding the reproducibility of crowd-reported security vulnerabilities. In: Proceedings of the USENIX Security Symposium, pp. 919–936 (2018)
18. OpenAI: GPT4-Turbo. https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4
19. OpenAI: OpenAI Platform. https://platform.openai.com/docs/overview

20. Peng, J., et al.: 1dVul: Discovering 1-day vulnerabilities through binary patches. In: Proceedings of the International Conference on Dependable Systems and Networks, pp. 605–616 (2019)
21. Xia, C.S., Paltenghi, M., Tian, J.L., Pradel, M., Zhang, L.: Fuzz4All: Universal fuzzing with large language models. In: Proceedings of the International Conference on Software Engineering (2024)
22. Yang, S., et al.: 1dFuzz: Reproduce 1-day vulnerabilities with directed differential fuzzing. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 867–879 (2023)
23. Zhang, J., Cui, Z., Chen, X., Yang, H., Zheng, L., Liu, J.: Cidfuzz: Fuzz testing for continuous integration. IET Software **17**(3), 301–315 (2023). https://doi.org/10.1049/sfw2.12125
24. Zhu, X., Böhme, M.: Regression greybox fuzzing. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 2169–2182 (2021). https://doi.org/10.1145/3460120.3484596