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 discuss how Large Language Models (LLMs) can be leveraged to generate inputs that trigger specific vulnerabilities. There are two main challenges: LLMs need to (i) correctly analyze the target vulnerability, and (ii) identify relevant fields to generate useful program inputs. We address these challenges by using a systematic approach with a three-stage prompting, where we provide necessary information at each stage and guide the LLM to ultimately generate an input for reproducing the target bug. With these LLM-generated program inputs, we perform directed fuzzing targeting the known vulnerabilities in real-world programs and show that our strategy can effectively generate useful inputs for vulnerability reproduction.

Keywords: bug reproduction, large language models, software security

1 Introduction

Analyzing known vulnerabilities, commonly referred to as 1-day vulnerabilities, is imperative in software security. By analyzing the issued patches for 1-day vulnerabilities, attackers can identify existing vulnerabilities and pose threats to software users who have not yet applied the patches [21]. Therefore, analyzing these vulnerabilities is crucial to quickly devise countermeasures.

However, 1-day vulnerability analysis is challenging because finding the program inputs that trigger the vulnerabilities requires considerable effort and time [16]. This process necessitates a high level of understanding of the program and input formats, and manipulating inputs to meet complex constraints demands substantial expertise.

Despite numerous studies on automated analysis of 1-day vulnerabilities, efficiency limitations still persist. Directed fuzzing [2,8,9] is a promising technique for reproducing a known vulnerability by gradually mutating the input to reach the target location. However, recent studies [10, 11] show that directed fuzzers suffer from quickly generating inputs for vulnerabilities that do not fit the heuristics they employ. Dynamic symbolic execution [4] and its variant, named directed

symbolic execution [13], are also effective in reproducing bugs, but they do not scale well for large programs due to the path explosion problem.

Therefore, we propose a novel technique based on large language models for an efficient and automatic 1-day vulnerability analysis. As large language models generate answers based on learned data, they do not rely on specific heuristics. Notably, they are capable of performing various tasks previously done by humans in a short time due to their extensive training on large datasets [3, 5].

However, there are several challenges in applying large language models for program input generation. First, the LLM may fail to understand the target vulnerability and its root cause, thus generating irrelevant inputs. Furthermore, even if the model understands the vulnerability, it may not be able to correctly identify which input fields are related to the vulnerability.

This paper addresses the aforementioned challenges using a systematic three-stage prompting method (§3) along with fuzzing. The key intuition of our approach is to use a series of prompts that build on each other to guide the LLM in generating inputs that are close enough to the bug-triggering inputs, so that those inputs can be used as initial seeds for directed fuzzing. First, we provide the LLM with information regarding the target vulnerability to help understand the root cause of the bug. Then, we ask the LLM to identify the fields related to the vulnerability based on the information obtained from the previous stage. In the final stage, we provide the model with a small program input (§3.3) and instruct the model to generate the bug-triggering input by adding or modifying the fields identified in the previous stage. In this way, we break down the challenging problem of finding the bug-triggering input into multiple smaller problems, such that the LLM can progress in a step-wise fashion by combining the information given in the prompts with the answers from previous stages. The generated inputs are not necessarily the exact bug-triggering inputs due to the limitations of the LLMs, but we expect them to serve as good initial seeds for directed fuzzing.

To evaluate the effectiveness of the proposed method, we apply our technique to 15 real-world programs with known vulnerabilities. From this experiment, the LLM successfully analyzes the cause of vulnerabilities and identifies the related fields in 9 (60%) and 7 (46%) programs, respectively. Moreover, using the inputs modified by the LLM as initial seeds for directed fuzzing improved the fuzzing results in 41.7% of the programs while showing the same performance in 36.5% and worse performance in 21.8% of the programs. Additionally, we conducted various case studies to further analyze the results and investigate the limitations of our LLM-based strategy.

In summary, our contributions are as follows:

- We propose a novel strategy that uses LLMs to generate program inputs for bug reproduction.
- We address the challenges of applying LLMs in bug-triggering input generation through a systematic three-stage prompting procedure.
- We evaluate our strategy by generating inputs to reproduce the known vulnerabilities in real-world programs and using these LLM-generated inputs as

the initial seeds for directed fuzzing. We measure the performance improvement in fuzzing to show the effectiveness of our technique.
– We publicize our code, prompts, and experimental results in support of open science: (link will be provided upon acceptance).

2 Related Work

2.1 1-day Vulnerability Reproduction

There have been several studies on reproducing 1-day vulnerabilities from patched binaries. 1dVul [19] combines directed fuzzing and directed symbolic execution to effectively reach a target location in the binary. 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 (TCS). 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]. Since our study mainly focuses on generating seeds for fuzzing, our approach can be used in conjunction with these techniques to reproduce known vulnerabilities.

2.2 Input Generation with LLM

Recent research has been actively exploring the use of large language models to generate or modify program inputs for fuzzing. However, they primarily 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. Furthermore, existing studies use relatively simple prompts to generate inputs without considering the complexity of the vulnerabilities.

Asmita *et al.* [1] demonstrated the effectiveness of using LLMs to generate diverse initial seeds tailored to the target's requirements for testing embedded applications such as BusyBox. 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. In contrast, our study provides prompts with vulnerability-related information, such as CVE descriptions and patched source code.

ChatAFL [14] improved the performance of protocol fuzzing by having LLM create inputs that alter the initial input values and states of the server program. However, ChatAFL does not utilize the information provided in the prompts in subsequent prompts, which could cause the LLM to generate inaccurate or irrelevant responses.

Fuzz4All [20] utilized LLMs to generate and modify inputs for fuzzing systems that take various programming languages as inputs. Similarly to our technique,

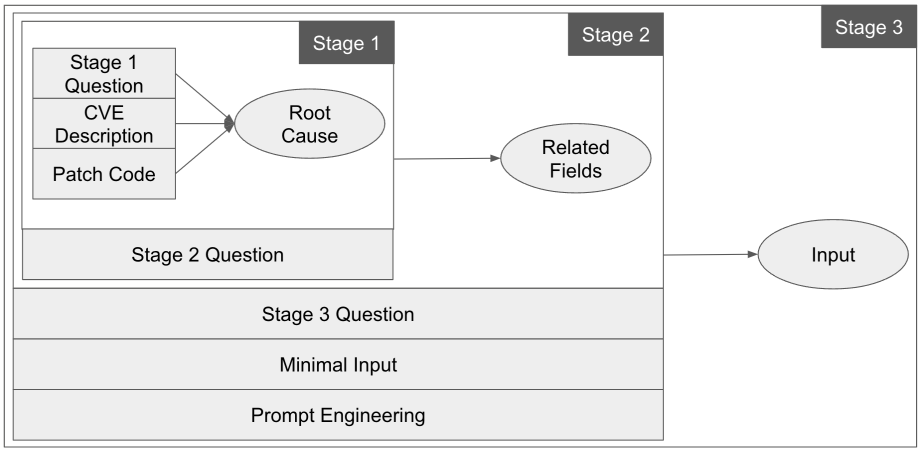

Fig. 1: Overview of 3-stage process.

they include official documentation and related code in their prompts. However, Fuzz4All relies on the LLM's learning capabilities to directly generate inputs without a staged approach, which could result in the model generating irrelevant responses. In contrast, our strategy employs a three-stage prompting method to sequentially extract essential information for generating inputs highly relevant to the vulnerabilities.

In summary, unlike the aforementioned studies, we use more complex prompts that include vulnerability-related information to enable the LLMs to generate more sophisticated and *directed* responses. Additionally, we conduct a sequential and systematic prompting process, extracting necessary information step-by-step to ultimately generate program inputs that are closely related to the vulnerabilities. To our knowledge, our study is the first to systematically utilize LLMs to generate inputs for reproducing known vulnerabilities.

3 Methodology

This section explains the core methodology of our research, which involves a three-stage prompt process and its specific design points. Our key insight is that we can generate inputs that are closely related to the vulnerabilities with continuous prompts. By attaining essential information for input generation at each stage and combining it with the information provided in previous prompts, we can guide the LLM to generate inputs that trigger the vulnerabilities.

Figure 1 illustrates the overall process of our research, and Figure 2 shows the specific template of our three-stage prompt. First, the Stage 1 prompt checks if the LLM can identify the root cause of the vulnerability. In Stage 2, the prompt determines whether the model can specify the fields related to invoking the vulnerability. In Stage 3, the prompt checks if the model can modify a minimal

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. 2: Template of prompts sent to LLM.

input to create an input that triggers the vulnerability. Detailed explanations of each stage prompt are provided in §3.1, §3.2, and §3.3. Finally, section §3.4 describes the implementation details of the prompt automation.

3.1 Stage 1 Prompt: 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 [15] for the vulnerability description.

Table 1: Minimal seeds for each program.

Program	**File Format**	**Size of Minimal Input (Byte)**
swftophp	SWF	15
lrzip	LRZ	46
xmllint	XML	36
cjpeg	BMP	58
cxxfilt	TXT	6
objcopy	ELF	460
readelf	ELF	460
pngimage	PNG	67
tiffcp	TIFF	63
sndfile-convert	WAV	45
openssl	DER	201
lua	LUA	2
php	JPEG	243
sqlite3	SQL	9
pdfimages	PDF	137

3.2 Stage 2 Prompt: Related Field Analysis

In the Stage 2 prompt, information obtained from the Stage 1 prompt and its response is used to ask the LLM to identify the related fields.

Providing Ground Truths The dotted boxes in Stages 2 and 3 of the prompts shown in Figure 2 represent the template of ground truths provided by our researchers when the model fails to generate accurate responses. In our methodology, the next stage's prompt uses the information from the previous prompts and answers. Therefore, even if the large language model returns inaccurate answers at each stage, it is necessary to provide ground truths to continue with the next prompt. For instance, if the model fails to identify the root cause of the vulnerability in Stage 1, our researchers provide the correct root cause in Stage 2. Similarly, if the model fails to find the related input fields in Stage 2, the correct fields identified by our researchers are provided in Stage 3.

3.3 Stage 3 Prompt: Input Generation

The Stage 3 prompt instructs the LLM to modify a minimal input using the information obtained in Stages 1 and 2 to create an input that triggers the vulnerability. Upon being modified, the modified input is used as the initial seed for directed fuzzing to verify if the input generated by the model helps in triggering the vulnerability.

Generating Minimal Inputs To find out if the large language model can generate inputs that trigger vulnerabilities by modifying the given inputs, we use minimal inputs in Stage 3. In this study, we define a minimal input, as the smallest input that allows the program to run without any warnings or errors. For example, in the case of the `tiffcp` program that accepts TIFF files, the minimal input would include the Image File Header (IFH) and four necessary entries in the Image File Directory (IFD).

Minimal inputs were created for each input format used in the vulnerabilities. To create the minimal inputs, we first identified the necessary fields required for the program to run normally by analyzing the source code of the programs. Then, we 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 sizes of the minimal inputs are summarized in Table 1.

Prompt Engineering Given the nature of security-related prompts, there were cases where the large language model refused to generate inputs in the third stage due to ethical concerns. To address this, we employed prompt engineering in Stage 3 to guide the model to modify the minimal input, as shown in Figure 2.

3.4 Prompt Automation

To automate the prompts, we used the Chat Completions API [18] provided by OpenAI to send prompts to the LLM. Using the templated prompts shown in Figure 2, vulnerability-specific information was added and sent automatically. After querying each stage, the correctness of the answers generated by the LLM, as determined by the researchers, was used to decide whether to provide the ground truths in the next stage prompt. Additionally, the prompts and responses from the previous stages were saved and sent to the model along with the next stage's prompt.

4 Evaluation

In this section, we evaluate our methodology to answer the following research questions.

RQ1. Can an LLM determine the root cause of the vulnerability?
RQ2. Can an LLM identify the input fields related to the vulnerability?
RQ3. Can an LLM modify minimal inputs?
RQ4. How effective are the modified inputs at triggering vulnerabilities?

Table 2: Benchmark programs.

Project	Program	Version	CVE	Type
Ming	swftophp	0.4.7	2016-9827	BOF
Lrzip	lrzip	0.631	2018-11496	UAF
Libxml2	xmllint	2.9.4	2017-9047	BOF
Libjpeg	cjpeg	1.5.90	2018-14498	BOF
Binutils	cxxfilt	2.6	2016-4487	ND
Binutils	objcopy	2.8	2017-8393	BOF
Binutils	readelf	2.9	2017-16828	IO
Libpng	pngimage	1.6.35	2018-13785	IO
LibTIFF	tiffcp	4.0.7	2016-10269	BOF
libsndfile	sndfile-convert	1.0.28	2018-19758	BOF
OpenSSL	openssl	1.1.0c	2017-3735	BOF
Lua	lua	5.4.0	2020-24370	IO
PHP	php	7.3.6	2019-11041	BOF
SQLite	sqlite3	3.30.1	2019-19923	ND
Poppler	pdfimages	0.73.0	2019-7310	BOF

4.1 Evaluation Setup

We selected 15 open-source programs that receive 14 different input formats as the benchmark for our evaluation. The programs were chosen from a commonly used fuzzing benchmark [7] and benchmarks from previous directed fuzzing papers [2, 8, 9]. We randomly selected one vulnerability for each program, resulting in a total of 15 vulnerabilities for the experiment. Detailed information about the benchmark is summarized in Table 2. We used GPT-4 Turbo [17] as the large language model. 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. Since the root causes of the vulnerabilities were made up of sentences, it was necessary to understand the responses before judging their success. In such cases, the responses generated by the LLM were evaluated based on the subjective understanding of the researchers. For related fields, the success was objectively determined by checking if the fields generated by the LLM matched those identified by our researchers.

Considering that the large language model often does not modify the minimal input in Stage 3 due to 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, our researchers applied the partial modifications to the minimal inputs manually, completing 7 and 10 initial seeds for fuzzing, respectively.

Table 3: Result of each stages.

Programs	Stage 1 (Success Rate)	Stage 2 (Success Rate)	Stage 3 (No. of Prompts)
swftophp	3 / 10	10 / 10	43
lrzip	0 / 10	0 / 10	10
xmllint	10 / 10	9 / 10	12
cjpeg	6 / 10	0 / 10	10
cxxfilt	0 / 10	0 / 10	13
objcopy	10 / 10	9 / 10	100
readelf	0 / 10	0 / 10	100
pngimage	10 / 10	6 / 10	17
tiffcp	0 / 10	0 / 10	24
sndfile-convert	0 / 10	0 / 10	21
openssl	10 / 10	10 / 10	100
lua	10 / 10	10 / 10	15
php	10 / 10	0 / 10	16
sqlite3	10 / 10	10 / 10	10
pdfimages	0 / 10	0 / 10	10
Total	9 / 15 (60%)	7 / 15 (46.7%)	12 / 15 (80%)

To evaluate the generated inputs, we used SelectFuzz (commit 6da35e0d) [12] as a directed fuzzer and AFL++ (v4.07c) [6] as an undirected fuzzer. For each program, 5 out of the 10 modified inputs were selected and fuzzed five times for 24 hours each. The selected inputs included those with the highest and lowest code coverage and three randomly chosen inputs. We ran the experiments on the 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.

4.2 Root Cause Analysis

Table 3 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 9 programs (60%). 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. With further advancements in LLMs, it is expected that vulnerability analysis will become more accurate.

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 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 model. As a result, the model 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 LLMs.

4.3 Related Fields Identification

In the second stage, 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 pre-learned data and 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 responses that LLM failed in this stage, we found that the model 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 model 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 model frequently failed to identify all the relevant fields. For CVE-2016-10269, which required three fields to trigger the vulnerability, the model only identified one of them. This shows that the model has difficulties in matching variables in the code to input format fields and identifying multiple related fields.

4.4 Modification of Minimal Inputs

In the third stage, the LLM successfully generated 10 modified inputs for 12 out of 15 programs (80%). However, despite prompt engineering efforts (§3.3), there were cases where the model refused to modify the minimal inputs due to ethical concerns. For instance, for CVE-2017-3735, which takes DER files as input format, only one modified input was generated after 100 prompt attempts. Similarly, for vulnerabilities with ELF inputs, no modified inputs were returned 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 4, and Table 5 reveal the fuzzing results using inputs generated by the LLM as initial seeds. The numbers represent the median time taken to find the

Table 4: Fuzzing results of AFL++.

Program	Minimal Input	LLM-generated Inputs				
		Best Coverage	Worst Coverage	Random 1	Random 2	Random 3
swftophp	13 (5)	14 (5)	50 (5)	15 (5)	7 (5)	21 (5)
lrzip	18 (5)	4527 (5)	2569 (5)	1964 (5)	4799 (5)	4513 (5)
xmllint	N.A. (0)	N.A. (0)	N.A. (0)	N.A. (0)	65393 (4)	N.A. (1)
cjpeg	80769 (3)	5487 (4)	N.A. (1)	25 (5)	34405 (4)	7175 (5)
cxxfilt	1108 (5)	581 (5)	508 (5)	172 (5)	662 (5)	895 (5)
objcopy	1005 (5)	1233 (5)	N.A. (1)	N.A. (2)	N.A. (2)	N.A. (0)
readelf	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)
tiffcp	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)
openssl	N.A. (0)	N.A. (0)	-	-	-	-
lua	N.A. (0)	13558 (5)	N.A. (0)	44 (5)	174 (5)	0 (5)
php	2839 (5)	9873 (5)	N.A. (2)	N.A. (1)	30820 (3)	13772 (4)
sqlite3	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)

Table 5: Fuzzing results of SelectFuzz.

Program	Minimal Input	LLM-generated Inputs				
		Best Coverage	Worst Coverage	Random 1	Random 2	Random 3
swftophp	45 (5)	13 (5)	983 (5)	50 (5)	27 (5)	20 (5)
lrzip	15 (5)	7065 (5)	20796 (5)	2043 (5)	3327 (5)	5923 (5)
xmllint	N.A. (0)	N.A. (1)	N.A. (0)	N.A. (0)	N.A. (1)	N.A. (0)
cjpeg	32520 (3)	100 (5)	N.A. (0)	37 (5)	187 (5)	704 (5)
cxxfilt	695 (5)	649 (5)	1242 (5)	498 (5)	1048 (5)	514 (5)
objcopy	1392 (5)	3006 (5)	N.A. (1)	N.A. (1)	N.A. (0)	N.A. (0)
readelf	-	-	-	-	-	-
pngimage	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)
sndfile-convert	N.A. (0)	N.A. (0)	N.A. (0)	N.A. (0)	354 (5)	N.A. (0)
openssl	-	-	-	-	-	-
lua	N.A. (0)	1099 (5)	N.A. (0)	N.A. (1)	N.A. (1)	0 (5)
php	-	-	-	-	-	-
sqlite3	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)

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 results marked with a dash (-) indicate that the fuzzer failed to perform static analysis which made fuzzing infeasible. Specifically, SelectFuzz failed for `readelf`, `openssl`, and `php` programs.

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

When fuzzing with the generated inputs by LLM as initial seeds, AFL++ and SelectFuzz succeeded in reproducing the vulnerability faster for 33% (5/15)

and 50% (6/12) of the programs, respectively, compared to using minimal input 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 model 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.

4.6 Case Studies

We conducted an in-depth analysis of two programs, `cjpeg` and `objcopy`, which showed good and bad fuzzing results. Listing 1.1 shows a partial source code related to CVE-2018-14498. This vulnerability occurs due to reading beyond the allocated memory of the colormap buffer at line 40. Specifically, the vulnerability arises when the pixel data value t, read from the input, exceeds the maximum index of the colormap buffer. As shown in lines 5-7, this maximum index is determined by the biClrUsed, which is also read from the input. From line 18, since the biBitCount field from the input BMP must be 8 to enter the vulnerable function get_8bit_row, the related fields for this vulnerability are biBitCount, biClrUsed, and the pixel data.

By analyzing the cjpeg-random-1 input, which showed good performance, we found that the LLM modified the biBitCount value from 24 in the minimal input to 8, allowing the execution to enter the vulnerable function. However, the LLM did not correctly identify the location of the biClrUsed field in the minimal input and changed other fields, causing incorrect values to be used as biClrUsed and pixel data. Despite this, the fuzzers managed to find an input that triggered the vulnerability within a short time by mutating the seed that enter the function where the vulnerability exists.

Listing 1.2 shows part of the code related to the `objcopy` vulnerability. This vulnerability arises from the incorrect assumption that section headers of type SHT_REL or SHT_RELA in ELF files start with the names ".rel" or ".rela". The code increases the name pointer, which points to the section header's name, by 4 or 5 and then uses the strcmp function to compare it with ".plt" at line 16. If the section name is shorter than 4 or 5 characters, there exist a scenario where the code reads memory beyond the allocated space for the section name. Therefore, the fields closely related to this vulnerability are sh_type and name.

```
METHODDEF(void) start_input_bmp(
    j_compress_ptr cinfo,
    cjpeg_source_ptr sinfo) {

  source->colormap = (*cinfo->mem->alloc_sarray)(
      (j_common_ptr) cinfo, JPOOL_IMAGE,
      (JDIMENSION) biClrUsed, (JDIMENSION) 3);

  ... // omitted
}

METHODDEF(JDIMENSION) preload_image(
    j_compress_ptr cinfo,
    cjpeg_source_ptr sinfo) {

  ... // omitted

  switch (source->bits_per_pixel) {
    case 8:
      source->pub.get_pixel_rows = get_8bit_row;
      break;
    case 24:
      ... // omitted
    case 32:
      ... // omitted
    default:
      ... // omitted
  }
  return (*source->pub.get_pixel_rows)(cinfo, sinfo);
}

METHODDEF(JDIMENSION) get_8bit_row(
    j_compress_ptr cinfo,
    cjpeg_source_ptr sinfo) {

  ... // omitted

  for (col = cinfo->image_width; col > 0; col--) {
    t = GETJSAMPLE(*inptr++);
    outptr[rindex] = colormap[0][t];  // crash!
    outptr[gindex] = colormap[1][t];
    outptr[bindex] = colormap[2][t];
    outptr += ps;
  }

  ... // omitted
  return 1;
}
```

Listing 1.1: Case Study of CVE-2018-14498.

```
asection* _bfd_elf_get_reloc_section (asection* reloc_sec) {

  ... // omitted

  type = elf_section_data(reloc_sec)->this_hdr.sh_type;
  if (type != SHT_REL && type != SHT_RELA)
    return NULL;

  name = reloc_sec->name;
  if (type == SHT_REL)
    name += 4;
  else
    name += 5;

  if (get_elf_backend_data(abfd)->want_got_plt
      && strcmp(name, ".plt") == 0) {  // crash!
      ... // omitted
    }

  ... // omitted
  return reloc_sec;
}
```

Listing 1.2: Case Study of CVE-2017-8393.

From analyzing the poorly performing objcopy-random-3 input, it was found that the model appended hexadecimal values representing ".unexpected" to the end of the .shstrtab section header data, which stores the names of the ELF file's section headers. While it is encouraging that the LLM identified the location of the section headers' names, the added string was too long to trigger the vulnerability. Additionally, while it was necessary to add a section header with the new name and set its type to SHT_REL or SHT_RELA, this was not done by LLM. Crucially, the new name data was added at the location before the start of the section headers, causing the e_shoff field, which indicates the start of the section headers, to have an incorrect value. This made generating an input to trigger the vulnerability infeasible.

5 Conclusion

This paper discusses how LLMs can assist in generating inputs that trigger specific vulnerabilities. Through a three-stage prompt process, we explored whether the model could infer the root causes and the related fields of known vulnerabilities. Additionally, given the root causes and related fields of the vulnerabilities, we prompted the model to modify minimal inputs into inputs that trigger the vulnerabilities and experimentally validated this using directed fuzzing. The evaluation results showed that the inputs generated by the LLM could reproduce the

bug faster for more than 41% of the programs on average, benefitting from its understanding of the vulnerabilities at each stage. 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.

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., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: 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. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. In: arXiv preprint arXiv:2107.03374 (2021)
6. 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)
7. Hazimeh, A., Herrera, A., Payer, M.: Magma: A ground-truth fuzzing benchmark. Proceedings of the ACM on Measurement and Analysis of Computing Systems **4**(3), 1–29 (2020)
8. 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)
9. 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)
10. 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)

11. 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)
12. 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)
13. 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)
14. 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)
15. MITRE Corporation: CVE-MITRE. `https://cve.mitre.org`
16. Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., Wang, G.: Understanding the reproducibility of crowd-reported security vulnerabilities. In: Proceedings of the USENIX Security Symposium. pp. 919–936 (2018)
17. OpenAI: GPT4-Turbo. `https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4`
18. OpenAI: OpenAI Platform. `https://platform.openai.com/docs/overview`
19. Peng, J., Li, F., Liu, B., Xu, L., Liu, B., Chen, K., Huo, W.: 1dVul: Discovering 1-day vulnerabilities through binary patches. In: Proceedings of the International Conference on Dependable Systems and Networks. pp. 605–616 (2019)
20. 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)
21. Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., Liu, T.: Patch based vulnerability matching for binary programs. In: Proceedings of the International Symposium on Software Testing and Analysis. pp. 376–387 (2020)
22. Yang, S., He, Y., Chen, K., Ma, Z., Luo, X., Xie, Y., Chen, J., Zhang, C.: 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/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`