

# Prompt Fuzzing for Fuzz Driver Generation

Yunlong Lyu  
Tencent Security Big Data Lab  
yunlong.lyu97@gmail.com

Peng Chen  
Tencent Security Big Data Lab  
spinpx@gmail.com

Yuxuan Xie  
Tencent Security Big Data Lab  
yxie0812@gmail.com

Hao Chen  
University of California, Davis  
chen@ucdavis.edu

## ABSTRACT

Writing high-quality fuzz drivers is time-consuming and requires a deep understanding of the library. However, the performance of the state-of-the-art automatic fuzz driver generation techniques leaves a lot to be desired. Fuzz drivers, which are learned from consumer code, can reach deep states but are restricted to their external inputs. On the other hand, interpretative fuzzing can explore most APIs but requires numerous attempts in a vast search space. We propose PROMPTFUZZ, a coverage-guided fuzzer for prompt fuzzing that iteratively generates fuzz drivers to explore undiscovered library code. To explore API usage in fuzz drivers during prompt fuzzing, we proposed several key techniques: instructive program generation, erroneous program sanitization, coverage-guided prompt mutation, and constrained fuzzer fusion. We implemented PROMPTFUZZ and evaluated its effectiveness on 14 real-world libraries, comparing it against OSS-Fuzz and the state-of-the-art fuzz driver generation solution (i.e., Hopper). The experiment results demonstrate that the fuzz drivers generated by PROMPTFUZZ achieve higher branch coverage that is 1.61 times greater than that of OSS-Fuzz and 1.67 times greater than that of Hopper. In addition, the fuzz drivers generated by PROMPTFUZZ successfully detect 33 true bugs out of a total of 44 crashes, which were previously unknown, and 27 of these bugs have been confirmed by the respective communities.

## 1 INTRODUCTION

Fuzzing is currently playing a crucial role in ensuring software security and stability. One example of this is OSS-Fuzz [1], which deploys state-of-the-art fuzzers for open-source software. As of February 2023, OSS-Fuzz has identified and resolved over 8,900 vulnerabilities and 28,000 bugs across 850 projects [2]. While grey-box fuzzers [3–5] have been successful in identifying bugs, the impressive results achieved by OSS-Fuzz can be largely attributed to the significant effort put into integrating new projects by various contributors. When integrating a project for fuzzing, developers must select an appropriate fuzzer and write high-quality fuzz drivers. Fuzz drivers are essential because they are used for parsing inputs from fuzzers and invoking the code in the software under test. However, writing high-quality fuzz drivers is a challenging task that is both time-consuming and requires a deep understanding of the library. Consequently, manually crafted fuzz drivers can only cover a small portion of the software usage, making the code unable to be adequately tested through fuzzing [6, 7].

Instead of writing fuzz drivers manually, the automatic fuzz driver generation technique learns the usage of library API from either existing code or dynamic feedback [7–16]. FUDGE [8], FuzzGen [9], and Utopia [13] extract the code of API usage API usages from source code statically, while APICraft [11] and WINNIE [12] record the API call sequences from the execution traces of existing processes dynamically. However, the efficiency of this method is limited by consumer code and cannot explore potential valid usage that consumer code does not cover. Hopper [7] transforms the problem of library fuzzing into the problem of interpreter fuzzing and learns the valid API usage from dynamic feedback of API’s invoking. Though it can cover most API functions, finding useful API invoking sequences that reach deep states requires numerous attempts in a huge search space.

Large language models (LLMs) have demonstrated remarkable success in generating programming code, providing hope for efficiently exploring the wide range of API usage without relying on consumer code. These models, such as the GPT-series[17–20], are typically composed of billions of parameters and undergo training on extensive code corpora. As a result, they demonstrate excellent performance in producing code that aligns with user intentions. Although existing works [15, 16, 21] have attempted to use LLMs for generating fuzz drivers, their instructions for generating fuzz drivers are limited to specific scenarios. The generated fuzz drivers still suffer from the problem of diversity of API usage and tend to fail to cover the rarely used code or deep states.

To address the aforementioned challenges, we introduce PROMPTFUZZ, a coverage-guided fuzzer that iteratively generates fuzz drivers to explore undiscovered library code. PROMPTFUZZ comprises an LLM-based program generator and a runtime-based program error oracle. The core idea behind PROMPTFUZZ is simple: instruct LLMs to generate the desired fuzz drivers through coverage guidance and utilize the program error oracle to ensure the effectiveness. The workflow of PROMPTFUZZ can be summarized in four steps: 1) Prompt LLMs with crafted instructions to generate programs focusing on the provided library API combinations. 2) Sanitize the erroneous programs that fail to execute or trigger false positives easily. 3) Guide the mutation of the LLMs prompts with the feedback of generated programs’ code coverage. 4) Convert the arguments of library API calls inside programs to extend their fuzzing capabilities. Finally, these fuzz drivers are fused into a fuzz driver compatible with existing fuzzers.

We implemented PROMPTFUZZ and evaluated its effectiveness on 14 real-world libraries, comparing it against OSS-Fuzz and the state-of-the-art fuzz driver generation solution (i.e., Hopper[7]). The experiment results demonstrate that the fuzz drivers generated

arXiv:2312.17677v1 [cs.CR] 29 Dec 2023

```

1 #include <vpx/vp8dx.h>
2 #include <vpx/vp8cx.h>
3 #include <vpx/vpx_decoder.h>
4
5 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data
6     , size_t size) {
7     // Create the decoder configuration
8     vpx_codec_dec_cfg_t dec_cfg = {0};
9     ...
10    // Initialize the decoder
11    vpx_codec_ctx_t decoder;
12    vpx_codec_iface_t *decoder_iface = vpx_codec_vp8_dx()
13        ;
14    vpx_codec_err_t decoder_init_res =
15        vpx_codec_dec_init_ver(&decoder, decoder_iface,
16            &dec_cfg, 0, VPX_DECODER_ABI_VERSION);
17    if (decoder_init_res != VPX_CODEC_OK) {
18        return 0;
19    }
20    // Process the input data
21    vpx_codec_err_t decode_res = vpx_codec_decode(&
22        decoder, data, size, NULL, 0);
23    if (decode_res != VPX_CODEC_OK) {
24        vpx_codec_destroy(&decoder);
25        return 0;
26    }
27    // Get the decoded frame
28    vpx_image_t *frame = NULL;
29    vpx_codec_iter_t iter = NULL;
30    while ((frame = vpx_codec_get_frame(&decoder, &iter))
31        != NULL) {
32        // Process the frame
33        vpx_img_flip(frame);
34        ...
35    }
36    // Cleanup
37    vpx_codec_destroy(&decoder);
38    return 0;
39 }

```

Figure 1: An example of a fuzz driver intended for libvpx.

by PROMPTFUZZ achieve a branch coverage that is 1.61 times greater than that of OSS-Fuzz and 1.67 times greater than that of Hopper. In addition, the fuzz drivers generated by PROMPTFUZZ successfully detect 33 true bugs out of a total of 44 crashes, which were previously unknown, and 27 of these bugs have been confirmed by the respective communities. Furthermore, PROMPTFUZZ’s power schedule effectively guides LLMs to generate programs that explore deep library code in most libraries.

## 2 BACKGROUND

### 2.1 Library Fuzzing

Library fuzzing has become increasingly important due to the widespread use of libraries in software development. In contrast to command-line programs that accept bytes stream as input, libraries usually have multiple entries (i.e., APIs), each of which imposes

more stringent input format constraints compared to command-line programs. To leverage existing fuzzers such as AFL[3], Libfuzzer[22], Angora[5], etc, fuzz drivers are developed to serve as delegates. These drivers receive random bytes from the fuzzer and subsequently convert these bytes into well-structured arguments for the API functions.

Figure 1 is an example of a fuzz driver intended for *libvpx*. The fuzz driver performs three primary functions. ❶ It invokes the API functions in a proper order to emulate the video decoding process. To ensure proper termination, it includes logic for handling errors and reclaiming resources in case of trivial errors. ❷ It prepares the input arguments derived from the random bytes generated by the fuzzer. These arguments should be carefully crafted since they may have complex constraints. For instance, the last argument of `vpx_codec_dec_init_res` in line 12 only takes integers within a limited range (as indicated by the macros), and the third argument of `vpx_codec_decode` in line 17 should be the length of the second argument. ❸ It can exercise the API functions efficiently to reach as much code as possible. Starting from line 25, the loop continuously fetches frames from the decoder to feed the processing APIs in the loop body. In this way, it can maximize the throughput of the byte stream input.

To create a high-quality fuzz driver, it is crucial to not only adhere to the constraints of the target library but also comprehensively test its APIs to cover as much code as feasible. This demands a deep understanding of the target libraries, making the automatic generation of fuzz drivers a challenging task.

### 2.2 Large Language Model

Large Language Models (LLMs) are deep learning models with sophisticated architectures and a massive number of parameters, allowing them to acquire knowledge from vast amounts of textual data. GPT3[18], ChatGPT[23] and GPT4[20] are current representative examples of LLMs. LLMs are trained to predict the next word, denoted as  $w_{n+1}$ , given a sequence of words  $w_1, w_2, \dots, w_n$ , by maximizing the objective of the language model, as shown in Equation 1.

$$P(w_1, w_2, \dots, w_n) = \prod_{i=1}^n P(w_i | w_1, w_2, \dots, w_{i-1}) \quad (1)$$

During the inference process, these models generate the next token auto-regressively,  $w_{n+1}$ , based on the previous tokens  $w_1, w_2, \dots, w_n$ , using the model weights learned from their parameters. The initial tokens provided by users are referred to as a **prompt**. To ensure that LLMs produce output that adheres to users’ instructions and aligns with their intents, a series of LLMs [20, 23–26] are also trained with Reinforcement Learning from Human Feedback (RLHF) [19], such as ChatGPT and GPT4.

## 3 DESIGN

### 3.1 Overview

The core principle behind PROMPTFUZZ is the iterative construction of LLM prompts. These prompts are carefully crafted to enable the generation of diverse programs while simultaneously eliminating invalid ones through the use of runtime-based sanitization. Unlike grey-box fuzzers that focus on mutating input bytes of programs

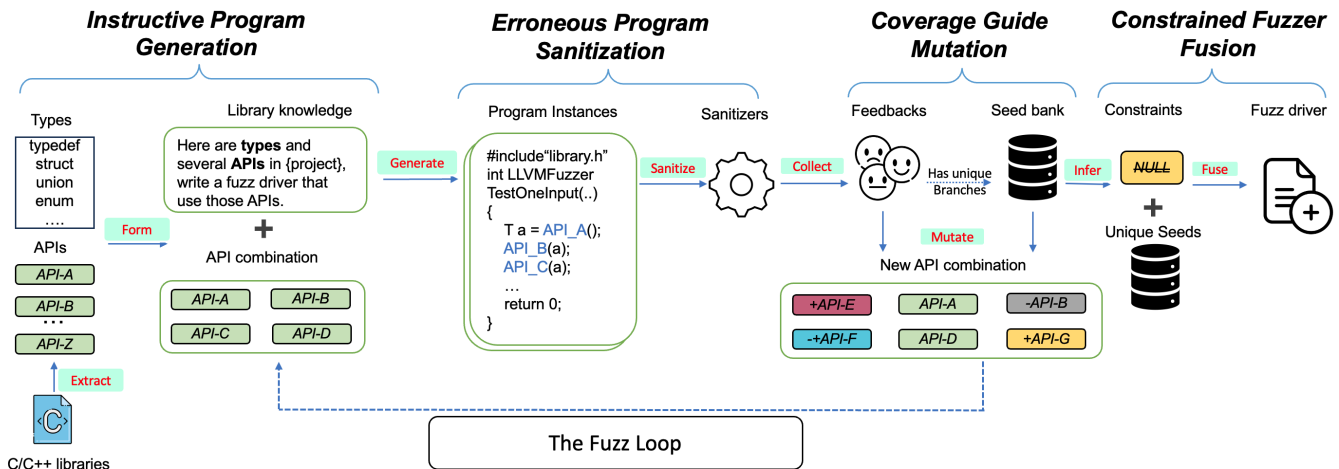


Figure 2: The workflow of fuzz driver generation in PROMPTFUZZ.

under test to reach deeper code, PROMPTFUZZ focuses on mutating the prompts of LLMs to produce programs that cover a broader range of library API utilization. The workflow of PROMPTFUZZ is depicted in Figure 2. Initially, PROMPTFUZZ extracts function signatures and type definitions from the headers of the libraries. These extracted details are then used to construct LLM prompts, instructing LLMs to generate programs that utilize the specified API combinations. The generated programs are then executed and sanitized based on their runtime behaviors. Code coverage is also collected during this sanitization process. Programs that pass the sanitization are stored in the *seed bank*, while those that trigger unique branches are marked as *unique seeds*. The code coverage of programs in the seed bank serves as feedback, guiding the mutation of the prompts. This feedback helps PROMPTFUZZ compose prompts with API combinations that are more likely to explore new code paths. This iterative process continues until either no new branches are discovered or the query budget is exhausted. In the final stage, PROMPTFUZZ infers the constraints imposed on library APIs within the seed programs. To enhance the fuzzing capabilities of these seed programs, PROMPTFUZZ converts the arguments with fixed values in *unique seeds* into arguments that can accept random bytes from fuzzers.

### 3.2 Instructive Program Generation

PROMPTFUZZ utilizes language models that have been trained on public code datasets and fine-tuned using RLHR[19] as the generator. Models such as ChatGPT[23] and GPT-4[20] are chosen for this purpose. Those LLMs possess the capability to generate code that follows programming syntax and semantics while also aligning with instructions. Although the generated programs may not always strictly adhere to the provided instructions, they still offer valuable exploration of different valid library usages. As a result, we can use instructions to guide LLMs in generating the desired programs that are effective for library fuzzing.

PROMPTFUZZ constructs LLM prompts that focus on generating specific combinations of library APIs. Figure 3 shows the prompt

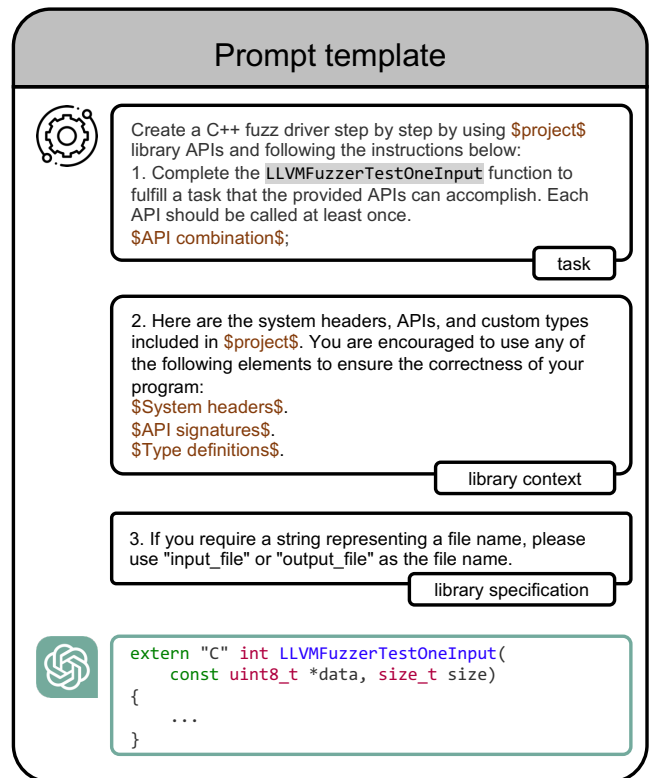
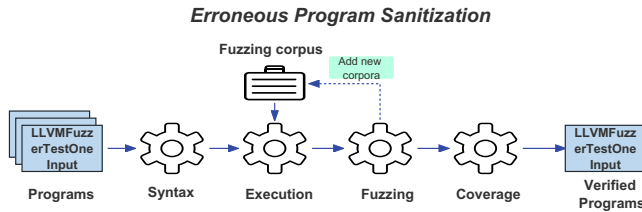


Figure 3: The prompt template designed in PROMPTFUZZ.

template used in PROMPTFUZZ. To compose diverse prompts for program generation, PROMPTFUZZ incorporates two crucial components: the *task* and the *library context*. Additionally, when specialized guidelines are required for creating fuzz drivers with specific libraries, the *library context* can be included. The *task* component specifies the details of the intended programs that LLMs should



**Figure 4: The workflow of erroneous program sanitization of PROMPTFUZZ.**

generate. It involves which API functions from the libraries should be used and these API functions should be included within a *LLVM-FuzzerTestOneInput* function. Furthermore, the component adopts a zero-shot Chain-of-Thought [27] to enhance the focus of LLMs. The *library context* component imparts a contextual understanding of library APIs to assist LLMs in accomplishing the work designated in *task*. This includes API signatures, custom type definitions, and included headers in the library. Given the context length limitations and the cost of corpora for current LLMs, PROMPTFUZZ limits the number of APIs and custom types used in the *library context*. When a library has an excessive number of API functions surpassing a set limit (for instance, 100), PROMPTFUZZ employs a random selection strategy to choose a manageable number of API functions each time, ensuring we stay within the limitation. As for custom types, PROMPTFUZZ is designed to select only those types that are utilized by the APIs in the chosen API combination, ensuring both relevance and efficiency. By integrating a contextual understanding of the libraries, the occurrence of "hallucination" code produced by LLMs can be significantly reduced[28–31]. The *library specification* plays a crucial role in guiding LLMs to generate code that adheres to specified patterns required by libraries. Some library APIs may read inputs from files, file streams, or file descriptors, which can deviate from the standard routines of fuzz drivers. By incorporating the relevant library specifications into the prompts for LLMs, we can make it easier to generate code patterns that adhere to these specifications. Once such a prompt is generated by filling the inputs in these components, PROMPTFUZZ queries LLMs with the prompt to generate programs.

### 3.3 Erroneous Program Sanitization

Due to the training data bias and imperfect code synthesis ability of LLMs, the code generated by LLMs is susceptible to be erroneous and insecure [32–34], and the accurate and correct generation of code by LLMs remains an unresolved challenge. As a result, the programs generated by LLMs are not suitable as direct targets for fuzzing. The presence of errors in these programs would make them difficult to fuzz or could overwhelm fuzzers with false warnings. According to several years of practice in fuzzing[21, 35], a program that is considered a good target for fuzzing should be, at the very least, free of any errors in the code itself. However, it is challenging to ensure that LLMs consistently generate error-free programs. To address this issue, PROMPTFUZZ employs a technique called erroneous program sanitization. This process sanitizes the problematic programs that contain identifiable code errors.

PROMPTFUZZ performs a four-step sanitization process for programs generated by LLMs, as shown in Figure 4. Firstly, it sanitizes programs that contain syntax errors identified by C/C++ compilers. Next, the remaining programs are compiled into executable files with multiple runtime sanitizers. These sanitizers capture the programs’ runtime behaviors and detect deviations from normal program behaviors. Subsequently, PROMPTFUZZ executes these programs using the provided corpus and excludes any programs that the sanitizers detect deviations in. Additionally, PROMPTFUZZ employs a fuzzing process to generate the inputs for the programs and preserves the inputs that trigger unique behaviors as part of the corpus. Following the fuzzing process, PROMPTFUZZ calculates the overall code coverage achieved by the program. Only programs that meet the code coverage criteria, indicating sufficient exercise of the library API functions, are considered to have passed the sanitization process.

**3.3.1 Execution Sanitization.** The potential buggy code patterns are scattered throughout the generated code, making it difficult to identify and address them. Additionally, detecting code logic errors in these programs remains an open question, further complicating the sanitization process.

Existing approaches, such as those mentioned in[36–45], either rely on extensive expert knowledge or suffer from precision issues. To ensure a convincing sanitization of programs generated by LLMs, we only exclude the programs that contain identifiable errors. To accomplish this, PROMPTFUZZ compiles the programs’ source code with several runtime error sanitizers, including Address-Sanitizer (ASan[46]), Undefined-Behavior-Sanitizer (UBSan[47]) and File-Sanitizer (see in Section 4.2). These sanitizers are equipped with predefined rules to detect any violations while designed with no expected false positives. The correctness of such a sanitization is based on two assumptions. **Firstly, if the library APIs are used correctly and provided with appropriate inputs in programs, they should not violate the sanitizers’ predefined rules. Secondly, if there is a misuse of library APIs, the sanitizers should be able to capture the abnormal runtime behaviors resulting from that misuse.** Built upon these assumptions, we execute the programs with the corpus generated in the fuzzing process (Section 3.3.2) to identify erroneous programs. Any violations reported by the sanitizers indicate deviations from correct library usage and the associated programs are therefore sanitized.

**3.3.2 Fuzzing Sanitization.** Initially, the program’s inputs used for sanitization can either be the seed inputs[48] provided by developers or just be empty. These seed inputs typically consist of a collection of representative inputs and are specifically designed for certain fuzz drivers, serving as a good starting point for fuzzing. However, as the programs are generated using different library APIs to accomplish various tasks, as explained in Section 3.2, each program may require custom inputs. Consequently, the initial seeds might not be suitable inputs for those programs, limiting the coverage of runtime behaviors that can be examined. The presence of unexamined program code poses a threat that can result in false negatives in the execution sanitization process.

To overcome the coverage limitations of the *execution sanitization*, PROMPTFUZZ conducts a fuzzing process to discover more unique inputs. These unique inputs are referred to as the fuzzing

corpus. Specifically, for a program that passed the previous sanitization processes, PROMPTFUZZ utilizes a grey-box fuzzer to mutate the inputs while simultaneously monitoring the code coverage of the program. If the code coverage has increased during each time interval (e.g., 60 seconds), PROMPTFUZZ continues the fuzzing process until it reaches the time budget (e.g., 600 seconds). Subsequently, PROMPTFUZZ adds the inputs that triggered new code coverage into the fuzzing corpus. The primary objective of our fuzzing process here is to generate the specific input required by each program, rather than identifying bugs within the library code. Although short-term fuzzing may not instantaneously produce the required program inputs, PROMPTFUZZ is designed to iteratively refine and evolve the fuzzing corpus over subsequent fuzzing rounds. This continuous evolution enhances the likelihood of generating suitable program inputs over time.

To show how the fuzzing process works, we take a program generated by LLMs shown in Figure 1 as an example. The program uses the `libvpx` APIs to decode a fragment of encoded video frames and iterates over each frame for processing. However, the fuzzing corpus of `libvpx` is a collection of video stream files (i.e., IVF) that contain both headers and video frames, making it unsuitable for the scenario depicted in Figure 1. When executing the program directly on the initial fuzzing corpus, the call to `vpx_codec_decode()` (line 17) returns a `VPX_CODEC_UNSUP_BITSTREAM` error, causing the program to exit immediately at line 20. Although these executions do not violate any rules of the runtime error sanitizers, the unexamined code (lines 23-31) poses a significant threat to the reliability of PROMPTFUZZ’s sanitization. With the help of an additional fuzzing process, PROMPTFUZZ successfully generates a suitable input for `vpx_codec_decode()` that passes the error checking (line 18) after multiple rounds of mutation of the initial fuzzing corpus.

**3.3.3 Coverage Sanitization.** After the previous sanitization steps, PROMPTFUZZ collects the code coverage of the remaining programs and finds the *critical paths* of the programs. The *critical paths* are the paths that contain the highest number of library API calls in the control flow graph of a program. These paths are of particular interest as they represent the API usage we want to test, rather than the error-handling code. In the case of the program depicted in Figure 1, the critical path executes lines 11, 12, 17, 25, 27, and 31. This path captures the essential API calls within the program. Following this analysis, PROMPTFUZZ sanitizes the programs where there is a library API call lying on the critical path that has not been executed. This ensures that the important API usage is thoroughly tested and validated.

The benefits of coverage-based sanitization are twofold. Firstly, previous runtime-based sanitization methods cannot accurately determine the correctness of unreachable code in programs. Although a fuzzing process is used to evolve the fuzzing corpus, it does not guarantee that all programs have been generated with suitable inputs. By employing coverage-based sanitization, we can effectively sanitize undecidable programs that contain unreachable code. While this approach may mistakenly exclude programs without misuse, it significantly reduces false positives in bug detection. Secondly, certain library API usages do not trigger abnormal behaviors that runtime sanitizers can capture. For instance, in line 17 of Figure 1, the code will never be executed if the call to `vpx_codec_dec_init()`

in line 12 is provided with incorrect arguments. *Coverage sanitization* can exclude the programs containing those silent library API misuses.

### 3.4 Coverage Guide Mutation

Upon the code synthesis capability of LLMs, a variety of programs could be generated from prompts crafted by PROMPTFUZZ. However, blindly assembling API combinations in the prompts would be inefficient. Thanks to code coverage as feedback, PROMPTFUZZ employs prompt-level power scheduling and mutation strategies to generate effective prompts.

**3.4.1 Power schedule.** PROMPTFUZZ maintains a set of visited branches and call graphs for the library under test. During each iteration of PROMPTFUZZ’s fuzzing process, it merges any newly discovered unique branches into the set, then calculates the branch coverage rate for each API, as shown in Equation 2. When computing the API’s branch coverage rate, it considers not only the branches within the function’s body but also the branches within the bodies of any recursive callees. For any API  $i$ , its new energy  $energy(i)$  is computed as Equation 3 shows:

$$cov(i) = \frac{\text{Number of Covered Branches Inside } i}{\text{Number of Total Branches Inside } i} \quad (2)$$

$$energy(i) = \frac{1 - cov(i)}{(1 + seed(i))^E \times (1 + prompt(i))^E} \quad (3)$$

where  $seed(i)$  is the count of seed programs called  $i$ ,  $prompt(i)$  is the count of  $i$  that has been selected in PROMPTFUZZ’s prompts and  $E$  is an exponent to regulate the importance of the frequency of  $i$ . The energy assigned to APIs is higher for those that are less exercised. The higher energy of an API indicates a higher likelihood of the API being selected in API combinations within prompts of LLMs.

**3.4.2 Mutation strategies.** PROMPTFUZZ focuses on mutating the API combinations in the prompts to instruct program generation. PROMPTFUZZ implements the following mutation operations on API combinations that are commonly used in traditional fuzzing[3, 15, 49–51]:

- **Insertion(C, A):** Insert API  $A$  into combination  $C$ .
- **Replacement(C, A, B):** Replace API  $A$  in combination  $C$  with API  $B$ .
- **Crossover(C, S):** Combine combination  $C$  and  $S$  to create a new combination.

Guided by the energy of APIs, PROMPTFUZZ can efficiently schedule the mutation to assemble combinations of APIs to reach previously unexplored code. However, combining APIs based solely on their degrees of exploration without considering their potential interrelationships hinders the ability of LLMs to generate programs that explore complex API relations. To facilitate more effective mutation, PROMPTFUZZ also collects statistics that reflect the effectiveness of API combinations used in the prompts. Specifically, PROMPTFUZZ collects the following statistics from the code of each program in the seed bank:

- **Density:** The maximum number of library API calls that exhibit explicit data dependence with each other.

- **Unique branches:** The number of unique branches triggered during program execution.

PROMPTFUZZ quantifies the quality of a program by using Equation 4, where the program with more correlating API calls and more discovering branches is assigned a higher quality.

$$\text{quality}(g) = \text{Density}(g) \times (1 + \text{Unique\_branches}(g)) \quad (4)$$

During each iteration of PROMPTFUZZ’s fuzzing process, PROMPTFUZZ sequentially explores the seed bank and updates the qualities of the *unique seeds*. Using the feedback from the energies of the library APIs and seed qualities, PROMPTFUZZ applies the algorithm outlined in Algorithm 1 to select new API combinations. If there are insufficient *unique seeds* in the current iteration, PROMPTFUZZ enters the warm-up stage (lines 3-7 in Algorithm 1). This stage randomly assembles API combinations by selecting APIs with high energies to broadly explore previously undiscovered library usage. In the mutation state (lines 9-23 in Algorithm 1), PROMPTFUZZ selects the library APIs called sequentially within a critical path of a *unique seed* as the pivot combination for mutation. APIs with no interactions with others are removed. The mutation centered around this pivot allows PROMPTFUZZ to deeply investigate intricate library usage. Upon obtaining a new API combination, it is used to construct the prompt for the next iteration of program generation.

---

**Algorithm 1** The strategy for selecting new API combinations.

---

```

1: function MUTATION(APIs, Seeds)
2:   Comb = {}
3:   if WarmUp(Seeds) then
4:     while len(Comb) < DefaultLen do
5:       A ← ChooseByEnergy(APIs)
6:       Insert(Comb, A)
7:     end while
8:   else
9:     seed ← ChooseByQuality(Seeds)
10:    Comb ← CriticalCalls(seed)
11:    mutator ← RandChoose(Insert, Replace, Crossover)
12:    if mutator is Insert then
13:      A ← ChooseByEnergy(APIs)
14:      Insert(Comb, A)
15:    else if mutator is Replace then
16:      A ← Choose(Comb)
17:      B ← ChooseByEnergy(APIs)
18:      Replce(Comb, A, B)
19:    else
20:      seedB ← ChooseByQuality(Seeds)
21:      CombB ← CriticalCalls(seedB)
22:      Comb = CrossOver(Comb, CombB)
23:    end if
24:  end if
25:  return Comb
26: end function

```

---

### 3.5 Constrained Fuzzer Fusion

After the fuzz loop is stopped, PROMPTFUZZ extends the fuzzing capabilities of the *unique seeds* and fuses them into a fuzz driver.

Firstly, PROMPTFUZZ infers the argument constraints imposed on library APIs from the programs stored in the seed bank. Then, PROMPTFUZZ transforms the code of *unique seeds* to replace the arguments with fixed values to receive random bytes from fuzzers without violating the imposed constraints. Finally, these programs are fused into a single fuzz driver in which the code of each *unique seed* is scheduled using random bytes.

**3.5.1 Argument Constraint Inference.** For arguments with immutable array type or scalar type, PROMPTFUZZ identifies them as potential recipients for random bytes from fuzzers. However, these arguments are commonly subjected to constraints that can significantly affect the effectiveness of the fuzz driver [7, 13]:

- **ArrayLength(A, n):** The argument  $n$  represents the length limit of array  $A$ .
- **ArrayIndex(A, i):** The argument  $i$  denotes an index within the array  $A$ .
- **FileName(S):** The argument  $S$  represents a string containing a file path.
- **FormatString(S):** The argument  $S$  represents a format string.
- **AllocSize(n):** The argument  $n$  indicates the size of buffer allocations.
- **FileDesc(fd):** The argument  $fd$  represents a file descriptor in the operating system.

PROMPTFUZZ deduces these constraints through static code analysis of programs in the seed bank. Specifically, PROMPTFUZZ infers *ArrayLength(A, n)* by checking for statements that indicate  $n$  as the size of  $A$ , such as `malloc`, `sizeof`, and `strlen`. Similarly, PROMPTFUZZ infers *FileDesc* by examining the data flow of the return value of file descriptor-related calls, such as `open` and `fileno`. The *ArrayIndex(A, i)* constraint is inferred by verifying that  $i$  is a scalar and its value is always smaller than the length of  $A$ . If an argument  $S$  is set to the string "input\_file" or "output\_file" (as specified in Figure 3), PROMPTFUZZ identifies a *FileName(S)* constraint on  $S$ . Similarly, if an argument  $S$  contains the character '%', PROMPTFUZZ identifies a *FormatString(S)* constraint on  $S$ . Because of the inconsistent statistical nature of the *AllocSize(n)* constraint, PROMPTFUZZ infers *AllocSize(n)* by varying the arguments of scalar type between their minimum and maximum values and executing them with the fuzzing corpus. If there is a significant difference in memory allocation, PROMPTFUZZ identifies corresponding *AllocSize(n)* constraint on the argument. In cases where multiple constraints are inferred on the same argument, the constraint that has been inferred the most times is adopted.

**3.5.2 Fuzz Driver Fusion.** Similar to FuzzedDataProvider[52], PROMPTFUZZ converts the random byte inputs from fuzzers into variables with designated types for library API calls. For each converted argument, PROMPTFUZZ attempts to provide them with several randomly generated values. If there was an error detected by PROMPTFUZZ’s sanitizers, the conversion of the argument would be canceled. Besides, the constant values of the converted arguments are collected as the initial seed corpora. For arguments with inferred constraints, their values are set to satisfy the identified constraints. Once the conversion is complete, PROMPTFUZZ synthesizes a new fuzz driver that conditionally calls each seed program. In this process, each

seed program is scheduled based on several specific bytes consumed from fuzzers.

## 4 IMPLEMENTATION

We implemented PROMPTFUZZ in 17,288 lines of Rust code. The following sections will introduce some essential components implemented in PROMPTFUZZ.

### 4.1 AST Visitor

PROMPTFUZZ parses the Abstract Syntax Tree (AST) of program code and utilizes the `clang_ast` crate[53] to deserialize the AST. Once the AST is deserialized in Rust, we implement an AST visitor to traverse the code’s ASTs and extract node attributes. This AST visitor enables PROMPTFUZZ to achieve argument constraint inference as discussed in Section 3.5.1. Additionally, PROMPTFUZZ performs source code transformation, as discussed in Section 3.5.2, by utilizing the source code locations embedded in the attributes of AST nodes. Building upon the AST visitor, we construct Control Flow Graphs (CFGs) for the programs and employ an intra-procedural data flow analysis engine on the CFG. The CFG allows us to analyze the critical path, while the data flow analysis engine assists in analyzing the dependency between library API calls.

### 4.2 File Sanitizer

In addition to the use of ASan and UBSan during the sanitization process of PROMPTFUZZ, we have also implemented a File-Sanitizer (FSan) to identify instances of error file operations, such as file descriptor leaks. These types of errors are often responsible for performance degradation but are not detectable by ASan or UBSan. Given that these errors significantly impact the effectiveness of the fuzz drivers generated by PROMPTFUZZ, we have implemented FSan to identify these issues. FSan achieves this by tracking the data flows of file descriptors, file streams, and file names, and by instrumenting detection code at the end of their lifespan within the source code.

## 5 EVALUATION

In this section, we conducted comprehensive evaluations to demonstrate the effectiveness of PROMPTFUZZ. Firstly, we evaluated PROMPTFUZZ on 14 widely-used open-source libraries that have undergone extensive fuzzing through OSS-Fuzz[1] over several years. We compared the code coverage achieved by PROMPTFUZZ’s fuzz drivers with that achieved by other approaches for fuzz driver generation. Secondly, we assessed the effectiveness of bug finding of the fuzz drivers generated by PROMPTFUZZ. Lastly, we evaluated the key components of PROMPTFUZZ to illustrate the contribution of each component to its overall effectiveness.

All experiments were conducted on a server with 48-core CPUs clocked at 2.50GHz and 128 GB of RAM, running the 64-bit version of Ubuntu 20.04 LTS. LibFuzzer[22] was the grey-box fuzz engine used in all evaluations.

### 5.1 Overall Results

We configured the `gpt-3.5-turbo-0613` and `gpt-3.5-turbo-16k-0613` models as the LLMs used for program generation. When a query’s tokens are shorter than the length limit of `gpt-3.5-turbo-0613`, we

choose `gpt-3.5-turbo-0613`; otherwise, we choose `gpt-3.5-turbo-16k-0613`, which comes at a higher cost but allows for a longer length limit. We set the temperature parameter to 0.9 for the LLMs and sampled 10 programs for each query. The default API combination length is chosen as 5. In the evaluations of PROMPTFUZZ, the fuzz loop was terminated after 10 consecutive iterations where no new branches were discovered, and the fused fuzz driver for each library was executed with a 24-hour timeout. To mitigate statistical errors, each experiment was repeated five times, and the average results were reported.

Under the experimental setup, we used PROMPTFUZZ to generate fuzz drivers for 14 open-source libraries and detect bugs. The results of these experiments are summarized in Table 1, which provides the statistics about the tested libraries, the generated fuzz drivers, branch coverage, and bug detection results. PROMPTFUZZ successfully generated 3,785 seed programs for these 14 libraries with the cost of \$63.14 for querying the LLMs (\$4.15 per library on average)<sup>1</sup>. Overall, the fuzz drivers generated by PROMPTFUZZ achieved a branch coverage of 40.12% on the tested libraries, which is 1.61x greater than OSS-Fuzz and 1.67x greater than Hopper, and detected 33 previously unknown bugs. All bugs found have been reported to the corresponding communities. In the following sections, we will detail the results of our evaluations.

### 5.2 Comparison with Hopper and OSS-Fuzz

To evaluate how effective the fuzz drivers generated by PROMPTFUZZ on code coverage are, we compare the branch coverage of libraries against the manually crafted fuzzers in OSS-Fuzz and the state-of-the-art automatic library fuzzing solution: Hopper. During the evaluation, we ran Hopper and fuzz drivers of OSS-Fuzz on each library for the same 24-hour period. If there are multiple fuzz drivers for a library in OSS-Fuzz, we ensured each driver ran independently on a distinct CPU core for the same 24-hour duration.

The evaluation results are shown in Table 1. In the comparison of PROMPTFUZZ against fuzz drivers from OSS-Fuzz and Hopper on 14 libraries, PROMPTFUZZ demonstrates the highest branch coverage in 8 out of the 14 libraries. Among the remaining 6 libraries where PROMPTFUZZ did not top the list, the coverage shortfall in `cJSON` and `zlib` was marginal. For `libjpeg-turbo`, `libpcap`, `re2`, and `c-ares`, the coverage gap was within a range of 1000 code branches, a margin that is considered acceptable within the scope of this study.

Compared to OSS-Fuzz, PROMPTFUZZ achieved higher branch coverage (40.07%) than OSS-Fuzz (24.88%) in all the libraries. The results become even more remarkable given the fact that multiple fuzz drivers built for `curl`, `zlib`, and `lcms` are provided in OSS-Fuzz and these libraries thus have been fuzzed for more than 24 hours. This higher branch coverage achieved by PROMPTFUZZ can be primarily attributed to its capability of generating programs that cover a wide range of library usage scenarios.

Compared to Hopper, which automatically synthesizes fuzz drivers through interpretative fuzzing, PROMPTFUZZ achieved higher total branch coverage (40.07% vs.24.05%) as well. There are two main reasons for PROMPTFUZZ’s better performance compared to Hopper.

<sup>1</sup>At the time of experiments, the input and output prices for `gpt-3.5-turbo-0613` are 0.0015 and 0.002 per thousand tokens respectively, and the prices for `gpt-3.5-turbo-16k-0613` are 0.003 and 0.004 per thousand tokens respectively.

Table 1: Overall results for PROMPTFUZZ-generated fuzz drivers

Tested Library					Generated Programs			Branch Coverage Comparison			Detected Bugs		
Name	Version	LoC	#APIs	#Branches	Total	#Seeds	Cost	PromptFuzz	OSS-Fuzz	Hopper	UC	VB	C
<b>curl</b>	8.4.0	154K	93	26,644	2,460	664(106)	\$3.58	<b>5,283(19.82%)</b>	#20 / 822(3.09%)	3,144(11.80%)	0	0	0
<b>libTIFF</b>	4.6.0	108K	195	14,204	2,110	153(71)	\$4.02	<b>7,448(52.43%)</b>	#1 / 5,740(40.60%)	4,283(30.29%)	6	6	6
<b>libjpeg-turbo</b>	3.0.1	144K	77	10,972	2,730	180(82)	\$4.98	5,186(47.26%)	#9 / <b>6,187(56.39%)</b>	4,148(37.81%)	3	2	2
<b>sqlite3</b>	3.43.2	413K	289	38,056	2,210	404(74)	\$2.90	<b>28,016(73.61%)</b>	#1 / 9,760(25.64%)	9,264(24.34%)	5	3	3
<b>libpcap</b>	1.10.4	58K	84	7,816	2,510	151(49)	\$3.68	2,974(39.25%)	#3 / 3,145(41.51%)	<b>3,784(48.40%)</b>	6	3	3
<b>cJSON</b>	1.7.16	10K	76	1,020	2,470	209(54)	\$3.40	846(82.94%)	#1 / 475(46.57%)	<b>881(86.37%)</b>	5	4	3
<b>libaom</b>	3.7.0	530K	47	61,702	2,840	237(59)	\$4.11	<b>15,811(25.62%)</b>	#1 / 10,984(18.01%)	8,425(13.65%)	3	3	3
<b>libvpx</b>	1.13.1	362K	40	35,544	3,430	396(98)	\$6.16	<b>7,434(20.91%)</b>	#2 / 4,721(13.32%)	3,232(9.10%)	4	4	4
<b>c-ares</b>	1.20.0	59K	61	4,038	1,590	126(38)	\$2.47	2,141(53.02%)	#2 / 791(22.80%)	<b>2,898(71.76%)</b>	3	2	2
<b>zlib</b>	1.3	30K	87	2,894	1,630	259(82)	\$2.41	2,210(76.36%)	#9 / 1,525(52.80%)	<b>2,247(77.64%)</b>	1	0	0
<b>re2</b>	bc0faab	28K	70	4,940	2,140	101(23)	\$3.30	3,192(64.61%)	#1 / <b>3,900(78.94%)</b>	3,273(66.25%)	0	0	0
<b>lcms</b>	2.15	45K	286	8,806	9,160	402(96)	\$14.04	<b>3,742(42.49%)</b>	#8 / 3,049(34.62%)	2,569(29.17%)	2	0	0
<b>libmagic</b>	FILE5_45	33K	18	7,440	2,010	217(32)	\$2.41	<b>4,697(63.67%)</b>	#3 / 4,628(62.74%)	3,885(52.67%)	4	4	1
<b>libpng</b>	1.6.40	57K	246	7,732	3,560	286(99)	\$5.68	<b>3,906(50.51%)</b>	#1 / 1,967(25.44%)	3,721(48.14%)	2	2	0
<b>Total</b>	-	2M	1,669	231,808	40,850	3,785(963)	\$63.14	<b>92,886(40.07%)</b>	57,694(24.88%)	55,754(24.05%)	<b>44</b>	<b>33</b>	<b>27</b>

Seeds = The number of seed programs present in the seed bank (and the count of *unique seeds* among them); UC = Number of reported unique crashes; VB = Number of valid bugs identified by manually review; C = Confirmed bugs after reported to the corresponding communities; The number of fuzzer drivers crafted for each library in OSS-Fuzz is prefixed with the '#' symbol.

Firstly, leveraging internal knowledge of LLMs, PROMPTFUZZ effectively extracts complex information about API interdependency in various library APIs such as *libTIFF*, *sqlite3*, and *lcms*, whereas Hopper blindly infers them. Secondly, Hopper was unable to generate the necessary code pattern for libraries that require iterative calls of library APIs, such as *libaom*, *libvpx*, and *re2*, because it lacked support for conditional grammars. In contrast, PROMPTFUZZ is capable of supporting all types of control flow transitions. Overall, PROMPTFUZZ generates fuzz drivers with higher overall code coverage than OSS-Fuzz and Hopper.

### 5.3 Effectiveness on Bug Detection

For the crashes reported by PROMPTFUZZ, we removed duplicate crashes by analyzing the call traces. In the end, the number of unique crashes reported was 44. We manually reviewed the code and documentation to verify the validity of these unique crashes. Out of the reported 44 unique crashes, 33 were identified as valid bugs and have been reported to the respective communities. At the time of writing, 27 of the identified bugs have been confirmed, while the remaining 6 bugs are awaiting responses. The details of those bugs are available at <https://github.com/PromptFuzz/PromptFuzz>.

For the remaining 11 crashes that were determined to be non-effective warnings, most of them resulted from dereferencing null pointers returned by library API calls. As demonstrated in Section 5.4.2, the conversion of arguments of library API calls significantly enhances the bug-finding capability of the fuzz drivers, but it also increases the likelihood of library APIs entering error states and returning null pointers. If the generated programs do not check every return value of library APIs, spurious crashes may occur when these null pointers are accessed by subsequent library API calls. It is important to note that we do not consider these crashes to be false positives in PROMPTFUZZ's bug detection. Instead, they are robustness issues stemming from the fact that the library APIs fail to check the passed null pointers. Excluding the 7 warnings reported due to library API robustness issues, **only 4 crashes were identified as false positives in PROMPTFUZZ's bug detection.**

**We argue that PROMPTFUZZ achieves a detection accuracy of 90.90% (40/44).** Among these 4 false positives, 2 crashes were caused by constraints that PROMPTFUZZ failed to infer from the libraries, while the other 2 are considered misuse of the target library that escaped program sanitization due to their complex triggering mechanisms.

### 5.4 Effectiveness of PROMPTFUZZ's components

In this section, we conducted experiments to investigate the impact of the proposed techniques on the effectiveness of PROMPTFUZZ. Table 2 presents the detailed analysis results for the sanitized erroneous programs and the inferred argument constraints of PROMPTFUZZ in previous experiments.

**5.4.1 Erroneous Programs Sanitization.** The numbers of programs sanitized by each process of PROMPTFUZZ's sanitization are shown in Table 2. It can be observed that the majority of the erroneous programs (23,821, 63.89%) were sanitized due to syntax errors. Additionally, the *execution* and *fuzzing* sanitization processes of PROMPTFUZZ identified 10,260 programs (27.51%) that exhibited abnormal runtime behaviors. Furthermore, 3,202 programs were sanitized by the *coverage* sanitization due to insufficient code coverage. Among the 10,260 programs sanitized by PROMPTFUZZ's *execution* and *fuzzing* sanitization, we analyzed their crash reports to investigate the factors contributing to the abnormal runtime behaviors. The most prevalent issues detected were segmentation violations (3,394, 33.07%) and memory leaks (3,003, 29.26%) identified by the sanitizers. Specifically, PROMPTFUZZ's FSan (detailed in Section 4.2) detected 324 programs that contained leaks of opened files.

To investigate whether the programs were correctly sanitized, we conducted a study in which we randomly selected 10 programs for each library that were sanitized by the *execution* and *fuzzing* sanitization and 10 programs that were sanitized by the *coverage* sanitization. We reviewed the code of these programs and conducted careful debugging to determine if they had been properly sanitized. The results revealed that almost all of the 140 programs

**Table 2: The results of sanitized erroneous programs and inferred argument constraints of PROMPTFUZZ.**

Library	Sanitized Erroneous Programs			Inferred Argument Constraints					
	Syntax	E & F	Coverage	ArrayLength	ArrayIndex	Format	FileName	FileDesc	AllocSize
curl	1289	472	123	13 / 13 / 0	0 / 0 / 0	10 / 5 / 0	2 / 2 / 1	3 / 3 / 0	0 / 0 / 0
libTIFF	1301	994	60	25 / 19 / 1	0 / 0 / 0	0 / 0 / 0	6 / 3 / 2	3 / 1 / 1	4 / 4 / 0
libjpeg-turbo	1948	267	335	25 / 23 / 3	0 / 0 / 2	0 / 0 / 0	8 / 8 / 0	0 / 0 / 0	12 / 10 / 0
sqlite3	920	638	248	25 / 10 / 0	0 / 0 / 0	7 / 3 / 0	12 / 2 / 0	0 / 0 / 0	4 / 0 / 0
libpcap	1232	583	614	3 / 3 / 0	0 / 0 / 0	0 / 0 / 0	5 / 5 / 1	0 / 0 / 2	0 / 0 / 0
cJSON	561	1630	279	7 / 7 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	1 / 1 / 0
libaom	1437	244	372	7 / 7 / 0	1 / 1 / 0	0 / 0 / 0	0 / 0 / 1	0 / 0 / 0	0 / 0 / 0
libvpx	1936	676	415	3 / 3 / 0	1 / 1 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0
c-ares	863	541	60	21 / 20 / 0	0 / 0 / 1	0 / 0 / 0	0 / 0 / 0	2 / 2 / 0	0 / 0 / 0
zlib	709	477	185	26 / 25 / 0	0 / 0 / 1	2 / 1 / 0	2 / 2 / 1	1 / 1 / 0	2 / 2 / 0
re2	1182	814	43	6 / 5 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 2	0 / 0 / 0
lcms	6627	2048	93	25 / 23 / 2	3 / 3 / 3	1 / 1 / 0	6 / 6 / 0	0 / 0 / 0	7 / 4 / 0
libmagic	1295	276	222	2 / 2 / 0	0 / 0 / 0	0 / 0 / 0	6 / 6 / 0	1 / 1 / 0	0 / 0 / 0
libpng	2521	600	153	16 / 7 / 0	0 / 0 / 1	0 / 0 / 0	1 / 1 / 0	0 / 0 / 0	4 / 4 / 0
<b>Total</b>	<b>23,821</b>	<b>10,260</b>	<b>3,202</b>	<b>197 / 167 / 5</b>	<b>5 / 5 / 8</b>	<b>20 / 10 / 0</b>	<b>48 / 35 / 6</b>	<b>10 / 8 / 5</b>	<b>34 / 25 / 0</b>

E & F: Execution and Fuzzing. The numbers separated by slashes represent the counts of the **ground truth constraints**, **PROMPTFUZZ correctly inferred constraints**, and **PROMPTFUZZ error-inferred constraints**, respectively.

sanitized by runtime sanitization indeed contained misuses of library APIs. The only exception is a latent resource leak detected by FSan in *libpcap*<sup>2</sup>, which is a genuine bug that originates from file descriptor leaks resulting from the mismatched resource allocation and deallocation between the functions `pcap_create` and `pcap_close`. Without FSan, such a hidden bug that resides within the most commonly used code pattern in *libpcap* would never have been uncovered. For the 140 programs sanitized by the *coverage* sanitization, 108 of them were confirmed to have erroneous library usage and were correctly sanitized due to the presence of unreachable library API calls. Among these, 25 were caused by incorrect library initialization, 40 were caused by the wrong API context, and 43 were due to invalid library API configurations. The remaining 32 programs were mistakenly sanitized because the fuzzer failed to generate input that can reach certain library API calls that are theoretically reachable within the time budget assigned for the *fuzzing* sanitization process of PROMPTFUZZ (see in Section 3.3.2).

It is important to note that although the sanitization processes implemented in PROMPTFUZZ may unintentionally exclude correctly functioning programs and genuine buggy programs, they have a significant impact in reducing the occurrence of false crashes during bug detection.

**5.4.2 Argument Constraint Inference.** In Section 3.5.1, we proposed the techniques to infer constraints imposed on arguments of library APIs, and convert the library API call arguments to receive random bytes from fuzzers. To assess the accuracy of the PROMPTFUZZ’s constraint inference, we inspected documents of the tested libraries to collect the ground truth of API argument constraints. As Table 2 shows, PROMPTFUZZ achieves 91.24% (250/274) precision and 79.61% (250/314) recall on the inference of argument constraints. The false positives mainly owing to the absence of argument identifiers in the declarations of library APIs. This deficiency hampers the ability of LLMs to comprehend the functionalities of these arguments, consequently leading to inaccurate library API usage generation. Noticeably, constraints inferred by PROMPTFUZZ is intended to limit the incorrect conversion of fixed API arguments. Therefore, false

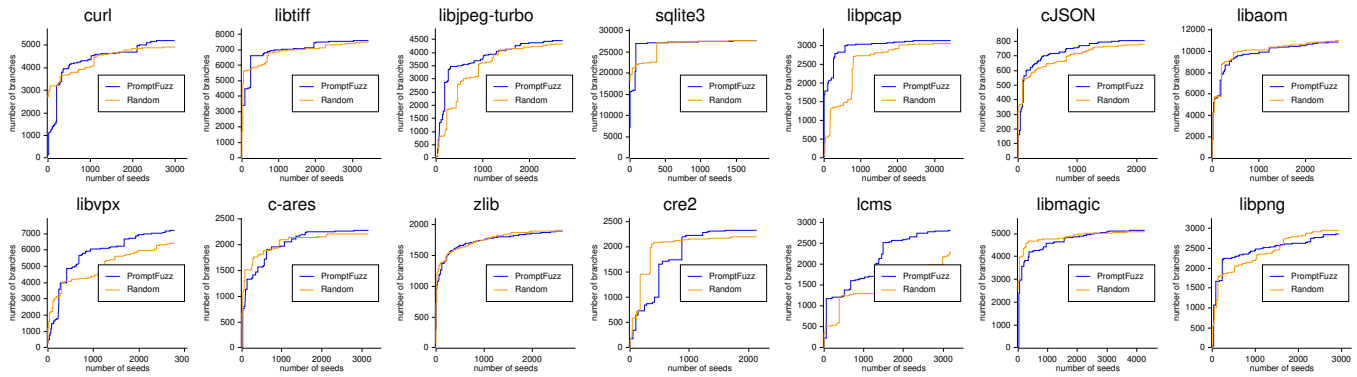
positives in inferred constraints will not cause additional spurious crashes. The false negatives were primarily because LLMs have not generated code for the relevant library APIs yet, and they rarely resulted in false positives in bug detection. Having those inferred constraints, PROMPTFUZZ can convert the library API arguments to receive random bytes without violating the constraints imposed by developers.

To quantify the number of bugs identified through the argument conversion of library API calls, we examined the program code to determine which arguments were responsible for the crashes. As a result, 15 out of the 33 identified bugs could only be detected using the additional converted arguments. For instance, all reported crashes in *libvpx* and *libaom* were triggered by incompatible flags and configurations passed to the codec. Without converting these arguments, the generated fuzz drivers would never have the chance to trigger them. These results highlight that, while the programs generated and sanitized by PROMPTFUZZ can serve as suitable targets for fuzzing, the ability of the resulting fuzz drivers to uncover new bugs is limited. PROMPTFUZZ’s capability to convert additional arguments of library APIs to receive random bytes from fuzzers significantly contributed to the discovery of new bugs.

**5.4.3 Coverage Guide Mutation.** PROMPTFUZZ develops a *coverage-guided mutation* to instruct LLMs in generating valuable programs. To evaluate its effectiveness, we experimented to compare it with a random blind mutation approach. In this experiment, the blind mutation approach was configured to randomly select library APIs with the same default combination length. To ensure fairness, both the coverage-guided mutation setup and the blind mutation setup were assigned the same query budget (i.e., \$5) and were executed until the budget was exhausted. Additionally, the temperature of the LLMs was set to 0.1 to reduce the randomness of the LLMs, and each experiment was repeated 5 times.

Figure 5 displays the accumulated covered branches attained by the generated seed programs during the fuzz loops of PROMPTFUZZ when configured with two different mutation methods. When giving the same query budget, the *coverage-guided mutation* outperformed random blind mutation in 11 out of the 14 libraries, with the

<sup>2</sup><https://github.com/the-tcpdump-group/libpcap/issues/1233>



**Figure 5: Numbers of branches covered by seed programs generated by PROMPTFUZZ under the setting of coverage-guided mutation and blind mutation.**

exceptions being *libaom*, *zlib*, and *libpng*. Despite the low growth rate of branch coverage in the warm-up stages, *coverage-guided mutation* surpassed random blind mutation in the 11 libraries due to the feedback obtained from the coverage and seed programs. This enabled PROMPTFUZZ to mutate prompts that incorporated meaningful combinations of APIs, resulting in the creation of programs that reached deeper library states.

The factors leading to the underperformance include the presence of loose coupling between APIs and the large number of APIs within these three libraries. In *libaom*, the APIs exhibit a high degree of coherence, and the interdependency between APIs is evident from their declarations. This clarity facilitates the generation of programs by LLMs, even when provided with randomly selected APIs. For *libpng*, the extensive number of APIs tend to trap the *coverage-guided mutation* setup in local states, while random mutation allows exploration of a broader range of APIs. This underperformance is expected to be resolved by allocating a larger query budget for LLMs. Although the *coverage-guided mutation* does not guarantee outperformance in all libraries, the experimental results demonstrate that it is the superior approach in most cases.

## 6 THREATS TO VALIDITY

We have made a diligent effort to ensure the correctness of the PROMPTFUZZ implementation. However, there may still be some remaining bugs in the implementation. Additionally, we have identified several threats to the validity of our experiments, apart from those posed by the implementation.

**LLM models.** In our experiments, we use GPT3.5[54] as the LLM model to generate programs with the prompts constructed by PROMPTFUZZ. The performance and behavior of different models and tools can vary significantly, leading to different outputs and results. Therefore, caution should be exercised when applying different LLM models on PROMPTFUZZ. Although we have conducted our experiments with a specific LLM model, our findings may offer insights into other LLM models, indicating that the performance of our tool may improve if more powerful LLM models are employed.

**Open-source libraries.** The evaluated libraries selected for our assessments are open-source. The source code of these libraries is publicly available and was used as the training data for our chosen

LLM model. If PROMPTFUZZ is applied to closed-source libraries, the evaluation results may differ. Nevertheless, fine-tuning LLMs could mitigate these issues.

## 7 RELATED WORK

### 7.1 Fuzz Driver Generation.

Several approaches have been proposed to facilitate the generation of fuzz drivers [7–16, 50, 55]. Fudge [8], FuzzGen [9], and Utopia [13] generate fuzz drivers by extracting library usage from consumer code. For instance, FuzzGen constructs an Abstract API Dependence Graph ( $A^2DG$ ) by analyzing the code of the Android Open Source Project (AOSP) and creates fuzz drivers by traversing the  $A^2DG$ . Meanwhile, APICraft [11] and Winnie [12] utilize the library usage learned from execution traces to create fuzz drivers. Unfortunately, these approaches fail to consider libraries without external consumers. To overcome this limitation, some approaches [10, 50, 55] have been proposed that generate fuzz drivers without relying on external consumers. GraphFuzz [50] relies on the library specification provided by users to compose fuzz drivers, while RULF [55] relies on strong type restraints in Rust to create fuzz drivers. However, these approaches either require human integration or are limited to domain-specific libraries. In addition to the aforementioned approaches, Hooper synthesizes fuzz drivers by fuzzing an interpreter to compose valid library API calls. However, the vast search space of API functions and arguments limits the effectiveness of fuzz drivers generated by Hooper. Additionally, TitanFuzz [15] and the method proposed by Google [16] rely on LLMs to generate fuzz drivers, but they do not consider their effectiveness. In comparison to these approaches, PROMPTFUZZ generates fuzz drivers without the requirement of external consumers and domain-specific models while maintaining their effectiveness in bug detection.

### 7.2 Deep Learning-based Software Testing.

Deep learning techniques are increasingly being utilized in software testing. SparrowHawk[56] and Goshawk[57] employ natural language processing (NLP) models to identify custom memory management functions within software projects, enhancing static code analysis. CarpetFuzz[6] utilizes NLP to extract API constraints from

software documents and detects violations by analyzing the dependencies between API calls. Pythia[58] is a grammar-based REST API fuzzer that utilizes a seq2seq model to achieve grammar mutation. In addition to these approaches that require training on specific deep learning models, there are several approaches designed directly on pre-trained LLMs [59–62]. Fuzz4All[60] and *Joshua et al* fuzz the program parser and language parser by utilizing LLMs to generate and mutate input to the parser. CodaMosa[59] employs LLMs to provide test cases for uncovered functions, addressing coverage plateaus caused by them. GPTFuzz[62] tests the robustness of deep learning library APIs by using LLMs to generate vulnerable cases. PROMPTFUZZ is a novel solution that aims for automatic fuzz driver generation. It constructs a fuzz loop to iteratively generate fuzz drivers that cover a broader range of library code. The fuzz drivers generated by PROMPTFUZZ can effectively test various library usage while maintaining a high detection accuracy.

## 8 CONCLUSION

This paper presents PROMPTFUZZ, a coverage-guided fuzzer for automatic fuzz driver generation. PromptFuzz generates fuzz drivers through prompt fuzzing, a novel fuzz loop built upon LLMs. Guided by coverage feedback, PROMPTFUZZ iteratively constructs prompts of LLMs to efficiently explore a wide range of API usage. Benefiting from the error program oracle we designed, PROMPTFUZZ can sanitize almost all erroneous programs generated by LLMs. By relying on the code synthesis capability of LLMs, PROMPTFUZZ creates fuzz drivers without the need for any consumer code or domain knowledge. The fuzz drivers generated by PROMPTFUZZ achieve higher branch coverage, 1.61 times greater than that of OSS-Fuzz and 1.67 times greater than that of Hopper. Additionally, the fuzz drivers generated by PROMPTFUZZ successfully detect 33 true bugs out of a total of 44 crashes, which were previously unknown, and 27 of these bugs have been confirmed by the community.

## REFERENCES

- [1] K. Serebryany, "OSS-Fuzz-google's continuous fuzzing service for open source software," in *Proceedings of the 26th USENIX Conference on Security Symposium (technical sessions)*. USENIX Association, 2017.
- [2] Taking the next step: Oss-fuzz in 2023. [Online]. Available: <https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html>
- [3] American fuzzy lop. [Online]. Available: <http://lcamtuf.coredump.cx/afl/>
- [4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, p. 1032–1043.
- [5] P. Chen and H. Chen, "Angora: efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, 5 2018.
- [6] D. Wang, Y. Li, Z. Zhang, and K. Chen, "Carpetfuzz: Automatic program option constraint extraction from documentation for fuzzing," in *Proceedings of the 32nd USENIX Conference on Security Symposium*. Anaheim, CA, USA: USENIX Association, 2023.
- [7] P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen, "Hopper: Interpretative fuzzing for libraries," in *ACM Conference on Computer and Communications Security (CCS)*, Copenhagen, Denmark, 2023.
- [8] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.
- [9] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287.
- [10] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "Intelligen: Automatic driver synthesis for fuzz testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 318–327.
- [11] C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "APICraft: Fuzz driver generation for closed-source SDK libraries," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2811–2828.
- [12] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.
- [13] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 746–762.
- [14] Y. Liu, Y. Wang, T. Bao, X. Jia, Z. Zhang, and P. Su, "Afgem: Whole-function fuzzing for applications and libraries," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 11–11.
- [15] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 423–435.
- [16] Fuzz target generation using llms. [Online]. Available: [https://google.github.io/oss-fuzz/research/llms/target\\_generation/](https://google.github.io/oss-fuzz/research/llms/target_generation/)
- [17] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [18] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, 2020, pp. 1877–1901.
- [19] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. F. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, 2022, pp. 27 730–27 744.
- [20] OpenAI, "Gpt-4 technical report," 2023.
- [21] C. Zhang, M. Bai, Y. Zheng, Y. Li, X. Xie, Y. Li, W. Ma, L. Sun, and Y. Liu, "Understanding large language model based fuzz driver generation," *arXiv preprint arXiv:2307.12469*, 2023.
- [22] libfuzzer – a library for coverage-guided fuzz testing. [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>
- [23] How to use chat-based language models. [Online]. Available: <https://platform.openai.com/docs/guides/chat/introduction>
- [24] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023.
- [25] Y. Sun, S. Wang, S. Feng, S. Ding, C. Pang, J. Shang, J. Liu, X. Chen, Y. Zhao, Y. Lu, W. Liu, Z. Wu, W. Gong, J. Liang, Z. Shang, P. Sun, W. Liu, X. Ouyang, D. Yu, H. Tian, H. Wu, and H. Wang, "Ernie 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation," 2021.
- [26] Google's bard. [Online]. Available: <https://bard.google.com/>
- [27] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Advances in Neural Information Processing Systems 35 (NIPS 2022)*, 2022.
- [28] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.
- [29] B. Bi, C. Wu, M. Yan, W. Wang, J. Xia, and C. Li, "Incorporating external knowledge into machine reading for generative question answering," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Nov. 2019, pp. 2521–2530.
- [30] B. Peng, M. Galley, P. He, H. Cheng, Y. Xie, Y. Hu, Q. Huang, L. Liden, Z. Yu, W. Chen, and J. Gao, "Check your facts and try again: Improving large language models with external knowledge and automated feedback," *arXiv preprint arXiv:2302.12813*, 2023.
- [31] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.
- [32] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

- [33] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32th USENIX Security Symposium (USENIX Security 23)*, 2023.
- [34] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," 2023.
- [35] What makes a good fuzz target. [Online]. Available: <https://github.com/google/fuzzing/blob/master/docs/good-fuzz-target.md>
- [36] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2019.
- [37] M. Wen, Y. Liu, R. Wu, X. Xie, S.-C. Cheung, and Z. Su, "Exposing library api misuses via mutation analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 866–877.
- [38] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1837–1852.
- [39] S. Nielebock, R. Heumüller, J. Krüger, and F. Ortmeier, "Cooperative api misuse detection using correction rules," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, p. 73–76.
- [40] H. Zeng, J. Chen, B. Shen, and H. Zhong, "Mining api constraints from library and client to detect api misuses," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2021, pp. 161–170.
- [41] M. Kechagia, X. Devroey, A. Panichella, G. Gousios, and A. van Deursen, "Effective and efficient api misuse detection via exception propagation and search-based testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, p. 192–203.
- [42] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.
- [43] H. J. Kang and D. Lo, "Active learning of discriminative subgraph patterns for api misuse detection," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2761–2783, 2021.
- [44] S. Nielebock, P. Blockhaus, J. Krüger, and F. Ortmeier, "Automated change rule inference for distance-based api misuse detection," *arXiv preprint arXiv:2207.06665*, 2022.
- [45] J. Yang, J. Ren, and W. Wu, "Api misuse detection method based on transformer," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 958–969.
- [46] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USENIX Association, 2012, p. 28.
- [47] Undefinedbehaviorsanitizer. [Online]. Available: <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>
- [48] How to prepare the seed corpus for oss-fuzz. [Online]. Available: <https://google.github.io/oss-fuzz/getting-started/new-project-guide/#seed-corpus>
- [49] Honggfuzz. [Online]. Available: <https://github.com/google/honggfuzz>
- [50] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1070–1081.
- [51] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.
- [52] Fuzzeddataprovider. [Online]. Available: <https://github.com/llvm-project/blob/main/compiler-rt/include/fuzzer/FuzzedDataProvider.h>
- [53] Clang ast deserializer in rust. [Online]. Available: <https://github.com/dtolnay/clang-ast>
- [54] Introducing chatgpt. [Online]. Available: <https://openai.com/blog/chatgpt>
- [55] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 581–592.
- [56] Y. Lyu, W. Gao, S. Ma, Q. Sun, and J. Li, "Sparrowhawk: Memory safety flaw detection via data-driven source code annotation," in *Information Security and Cryptology: 17th International Conference, Inscrypt 2021, Virtual Event, August 12–14, 2021, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 129–148. [Online]. Available: [https://doi.org/10.1007/978-3-030-88323-2\\_7](https://doi.org/10.1007/978-3-030-88323-2_7)
- [57] Y. Lyu, Y. Fang, Y. Zhang, Q. Sun, S. Ma, E. Bertino, K. Lu, and J. Li, "Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis," in *2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 1566–1566. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00137>
- [58] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," 2020.
- [59] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.
- [60] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," 2023.
- [61] J. Ackerman and G. Cybenko, "Large language models for fuzzing parsers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop*, ser. FUZZING 2023, 2023.
- [62] Y. Deng, C. Xia, C. Yang, S. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.