# Prompt Fuzzing for Fuzz Driver Generation

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

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

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

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

## Abstract

Crafting high-quality fuzz drivers not only is time-consuming but also requires a deep understanding of the library. However, the state-of-the-art automatic fuzz driver generation techniques fall short of expectations. While fuzz drivers derived from consumer code can reach deep states, they have limited coverage. Conversely, interpretative fuzzing can explore most API calls but requires numerous attempts within a large 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 propose several key techniques: instructive program generation, erroneous program validation, coverage-guided prompt mutation, and constrained fuzzer scheduling. We implemented PromptFuzz and evaluated it on 14 real-world libraries. Compared with OSS-Fuzz and Hopper (the state-of-the-art fuzz driver generation tool), fuzz drivers generated by PromptFuzz achieved 1.61 and 1.63 times higher branch coverage than those by OSS-Fuzz and Hopper, respectively. Moreover, the fuzz drivers generated by PromptFuzz detected 33 genuine, new bugs out of a total of 49 crashes, out of which 30 bugs have been confirmed by their respective communities.

## CCS Concepts

**• Security and privacy → Software security engineering**; **• Software and its engineering → Software testing and debugging**.

## Keywords

Fuzzing, Automated Test Generation, Vulnerability Detection

**ACM Reference Format:**
Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670396

## 1 Introduction

Fuzzing is crucial for software security and reliability. OSS-Fuzz [55], which deploys state-of-the-art fuzzers for open-source software, has identified and resolved over 8900 vulnerabilities and 28 000 bugs across 850 projects as of February 2023 [9]. These impressive results can largely be attributed to the significant efforts by the contributors to integrate new projects. When integrating a project for fuzzing, developers select an appropriate fuzzer and write high-quality fuzz drivers. Fuzz drivers are essential because they parse inputs from the fuzzers and invoke the code in the software under test. However, writing high-quality fuzz drivers is challenging because it is both time-consuming and requires a deep understanding of the library. Consequently, manually written fuzz drivers often invoke only a small portion of the software's functions and therefore limit the power of fuzz testing [13, 60].

Compared with manually written fuzz drivers, automatic techniques derive fuzz drivers by learning library API usage from either source code or runtime feedback [5, 13, 16, 17, 24, 25, 30, 38, 71, 72]. FUDGE [5], FuzzGen [24], and UTopia [25] extract the code of API usage from source code statically, while APICraft [71] and WINNIE [30] record the sequences of API calls from the execution traces of processes dynamically. However, since the traces contain only the API call sequences invoked by the consumer code, this method cannot learn valid API usage that is absent in the consumer code. Hopper, the state-of-the-art fuzz driver generation solution, transforms the problem of library fuzzing into the problem of interpretative fuzzing, which learns valid API usage from dynamic feedback of API invocations [13]. Although it can cover most API functions, it requires many attempts in a vast search space to find useful API invocation sequences that reach deep states.

Large language models (LLMs) have demonstrated remarkable success in generating program code, promising to effectively explore a wide range of API usage without relying on consumer code. Exemplified by the GPT-series [8, 49, 50, 53], they are trained on extensive code corpora and are able to produce code that aligns with user intentions. Although prior work [16, 17, 70] attempted to use LLMs for generating fuzz drivers, their instructions for generating fuzz drivers were limited to specific scenarios. As a result, the generated fuzz drivers suffered from low API usage diversity and failed to cover infrequently used code or deep states.

To address these challenges, we introduce PromptFuzz, a coverage-guided fuzzer that iteratively mutates prompts to explore undiscovered library code. Thanks to coverage guidance, the mutated prompts can direct LLMs to produce code that triggers complex scenarios or enters deep states. Since LLMs sometimes generate erroneous code, PromptFuzz employs multiple program error oracles for validating the generated code. The workflow of PromptFuzz is as follows: (1) Prompt LLMs with crafted instructions to generate programs focusing on the provided library API functions. (2) Eliminate erroneous programs which fail to execute or trigger false positives. (3) Guide the mutation of the LLMs prompts with the feedback of code coverage of the generated programs. (4) Convert the arguments of library API calls inside the generated programs from constants to variables whose assignments can be mutated during fuzzing. Finally, these fuzz drivers are fused into a fuzz driver compatible with existing fuzzers.

We implemented PromptFuzz and assessed it on 14 real-world libraries. Compared with OSS-Fuzz and Hopper [13], the fuzz drivers generated by PromptFuzz achieved 1.61 and 1.63 higher branch coverage than that by OSS-Fuzz and Hopper, respectively. Additionally, the fuzz drivers generated by PromptFuzz identified 33 genuine, new bugs from 49 crashes, out of which 30 bugs have been confirmed by their respective communities. Moreover, PromptFuzz's power scheduling 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. Unlike command-line programs, which process a byte stream as input, libraries possess multiple access points (i.e., API functions) that require more stringent inputs to adhere to strict format constraints. To leverage existing fuzzers [7, 11, 12, 40, 68], fuzz drivers are developed to serve as delegates. These drivers accept random bytes from the fuzzer and subsequently convert these bytes into well-structured arguments for API calls.

Figure 1 depicts a fuzz driver designed for *libvpx*. This driver fulfills three main functions: ❶ it properly invokes API functions to simulate the video decoding process. To ensure proper termination, the code of handling errors and reclaiming resources is incorporated to handle trivial errors; ❷ it constructs input arguments from randomly generated bytes, which must be carefully formatted due to their complex constraints. For example, the final parameter of `vpx_codec_dec_init_ver` (line 12) accepts only integers within a specified range, as defined by macros, and the third parameter of `vpx_codec_decode` (line 17) need to correspond to the length of the second parameter; ❸ it exercises the API functions to reach as much code as possible. Starting from line 25, the loop continuously fetches frames from the decoder and feeds them to the processing API functions in the loop body. In this way, it optimizes the throughput of the byte stream input.

To develop a high-quality fuzz driver, it is essential to adhere to the constraints of the target library and thoroughly test its API functions to achieve comprehensive code coverage. This requires a comprehensive understanding of the target libraries, rendering the automatic generation of fuzz drivers a challenging task.

```
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, size_t size) {
6      // Create the decoder configuration
7      vpx_codec_dec_cfg_t dec_cfg = {0};
8      ...
9      // Initialize the decoder
10     vpx_codec_ctx_t decoder;
11     vpx_codec_iface_t *decoder_iface = vpx_codec_vp8_dx();
12     vpx_codec_err_t decoder_init_res =
           vpx_codec_dec_init_ver(&decoder, decoder_iface,
           &dec_cfg, 0, VPX_DECODER_ABI_VERSION);
13     if (decoder_init_res != VPX_CODEC_OK) {
14         return 0;
15     }
16     // Process the input data
17     vpx_codec_err_t decode_res = vpx_codec_decode(&
           decoder, data, size, NULL, 0);
18     if (decode_res != VPX_CODEC_OK) {
19         vpx_codec_destroy(&decoder);
20         return 0;
21     }
22     // Get the decoded frame
23     vpx_image_t *img = NULL;
24     vpx_codec_iter_t iter = NULL;
25     while ((img = vpx_codec_get_frame(&decoder, &iter))
           != NULL) {
26         // Process the frame
27         vpx_img_flip(img);
28         ...
29     }
30     // Cleanup
31     vpx_codec_destroy(&decoder);
32     return 0;
33 }
```

**Figure 1: A fuzz driver for libvpx**

### 2.2 Large Language Model

Large Language Models (LLMs) are deep learning models with sophisticated architectures and numerous parameters, allowing them to acquire knowledge from vast amounts of textual data. GPT3 [8], ChatGPT [47] and GPT4 [49] are current representative examples of LLMs. LLMs are trained to predict the next word, denoted as $w_{n+1}$, given a sequence of words $w_1, w_2, ..., w_n$, by maximizing the objective function of the language model, as shown in Equation 1.

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

In the inference phase, LLMs auto-regressively generate the next token, $w_{n+1}$, based on prior tokens $w_1, w_2, ..., w_n$, utilizing the model weights learned from their extensive parameters. The starting tokens provided by users are known as a **prompt**. To ensure that LLMs produce output that is consistent with user instructions and aligns with their intents, a series of LLMs [19, 47, 49, 57, 59] have

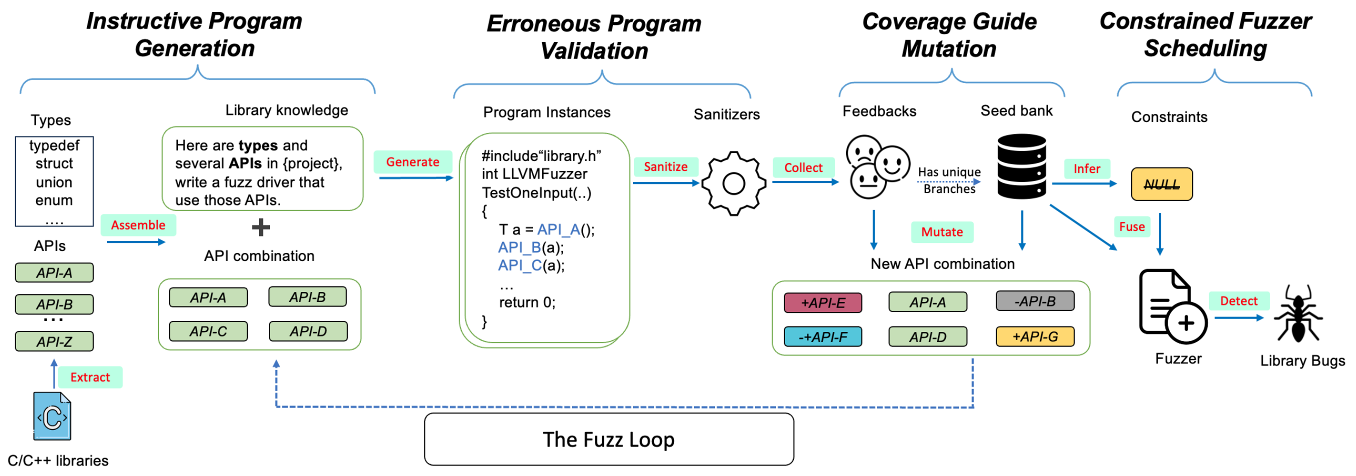


**Figure 2: Fuzz driver generation in PromptFuzz. *Seed* represents a program instance generated by LLMs.**

been enhanced with the training of Reinforcement Learning from Human Feedback (RLHF) [50], such as ChatGPT and GPT4.

## 2.3 LLM-based Fuzz Driver Generation

Very recently, there has been a growing interest in leveraging LLMs to enhance fuzzing tasks [3, 16, 17, 63, 73]. The primary challenges associated with using LLMs for fuzzing include automatic prompt construction and output validation.

In LLM-based fuzz driver generation, prompts usually consist of task descriptions and contextual information. To be as informative and instructive as possible, the task description should specify at least the target library and the API functions to be included for this round of code generation. Among the early attempts [15, 16, 63], researchers assign only one API for each prompt as the target. However, the code generated with such prompts tends to have the problem of being too simple or demonstrates rather common usage. In contrast, bugs usually exist in corner cases or complicated scenarios. On the other hand, the code generated by LLMs is hard to directly utilize for fuzzing as it is susceptible to be erroneous [36, 51, 54]. Current works [3, 17] rely on compilers or simple rules for output validation. However, these ways only report syntactic problems or shallow logic errors and fail to detect complex semantic errors (e.g., incorrect library usage). When utilized as fuzz drivers, such buggy code will incur many false positives.

## 3 Design

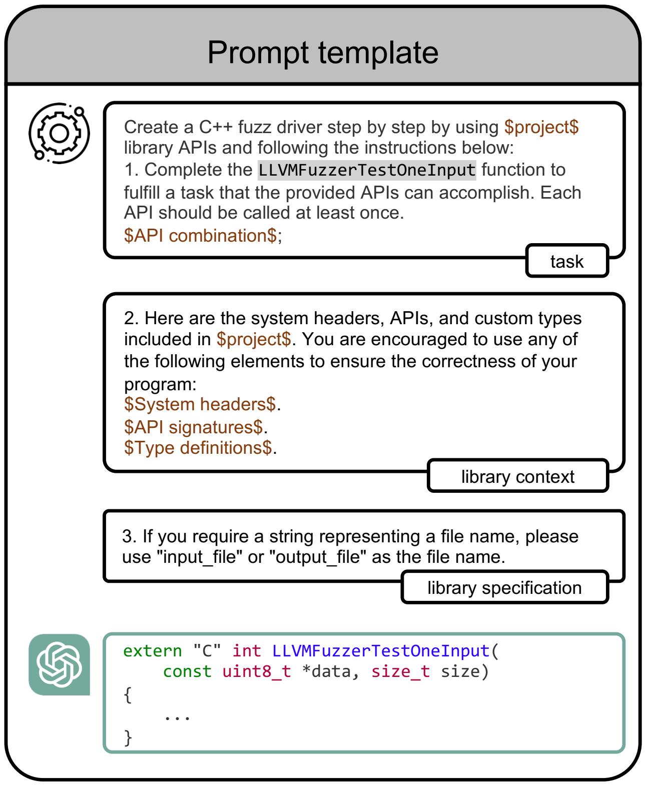
## 3.1 Overview

PromptFuzz generates high-quality fuzz drivers for effective library bug detection via coverage-guided LLM prompt construction. Unlike grey-box fuzzers, which mutate input bytes to reach deeper program code, PromptFuzz mutates LLM prompts to produce programs that cover a broader range of library API utilization. Initially, PromptFuzz constructs a prompt using randomly selected library API functions. Then, it mutates this prompt based on coverage feedback until the fuzzing reaches convergence for the target library. The mutations target the API functions within the prompts to generate diverse programs. Simultaneously, the generated programs are validated at runtime to ensure correctness. The workflow of PromptFuzz is depicted in Figure 2.

(1) PromptFuzz extracts function signatures and type definitions from the header files of a C/C++ library and uses them to construct prompts for instructing LLMs to generate programs that call these functions.
(2) PromptFuzz executes the generated programs, validates them based on their runtime behavior, and eliminates the erroneous ones. PromptFuzz also collects code coverage during the executions.
(3) PromptFuzz stores programs that pass the validation in a *seed bank*. It then uses their code coverage as feedback to mutate prompts towards API functions that are more likely to explore new code paths. This iterative process continues until PromptFuzz discovers no new paths or it exhausts the query budget.
(4) Finally, PromptFuzz infers the constraints imposed on library API functions within the seed programs. It converts the arguments of library API calls from constants, which LLMs generated, into variables that can take arbitrary values provided by the fuzzer while preserving the inferred constraints. To detect library bugs, PromptFuzz consolidates all the converted seed programs in a fuzz driver and then schedules each seed program to be fuzzed with random bytes from the fuzzers.

## 3.2 Instructive Program Generation

PromptFuzz instructs LLMs to generate the desired programs through zero-shot prompting [33]. It uses LLMs that have been trained on public code datasets and fine-tuned using RLHF [50] as the generator. Those LLMs possess the capability to generate code that both conforms to programming syntax and semantics and also aligns with instructions. We chose ChatGPT [47] and GPT-4 [49] as LLMs. While the generated programs may not always strictly


Prompt template

Create a C++ fuzz driver step by step by using $project$ library APIs and following the instructions below:
1. Complete the `LLVMFuzzerTestOneInput` function to fulfill a task that the provided APIs can accomplish. Each API should be called at least once.
$API combination$;

2. Here are the system headers, APIs, and custom types included in $project$. You are encouraged to use any of the following elements to ensure the correctness of your program:
$System headers$.
$API signatures$.
$Type definitions$.

3. If you require a string representing a file name, please use "input_file" or "output_file" as the file name.

```
extern "C" int LLVMFuzzerTestOneInput(
    const uint8_t *data, size_t size)
{
    ...
}
```

**Figure 3: Prompt template**

follow the instructions, they help explore valid library usage. Therefore, we use instructions in LLM prompts to steer LLMs toward generating desirable programs for library fuzzing.

PromptFuzz constructs prompts that instruct LLMs to generate programs with specific combinations of library API functions. To synthesize such an LLM prompt, PromptFuzz fills in a prompt template (Figure 3) with extracted library gadgets and a designated API combination. For effective fuzz driver generation, a template has the following components:

- The *task* component details the intended programs that LLMs should generate. It specifies which API functions from the libraries are mandatory within a *LLVMFuzzerTestOneInput* function.
- The *library context* includes API signatures, custom type definitions, and headers in the library. Given the context length limitations and token cost of current LLMs, PromptFuzz restricts the number of API functions and custom types in *library context*. When a library has too many API functions, exceeding 100 for instance, PromptFuzz uses a random selection strategy to choose a manageable subset each time. For custom types, PromptFuzz selects only those types that are used by the chosen API functions for relevance and efficiency. By integrating a contextual understanding of the libraries, we can significantly reduce the occurrence of "hallucination" code produced by LLMs [6, 26, 35, 52].

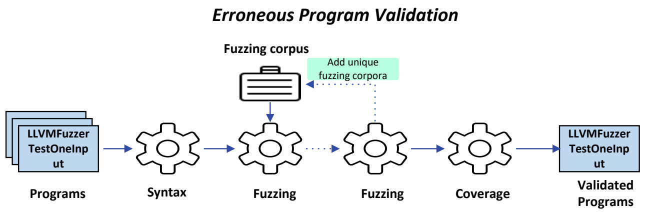


**Figure 4: Erroneous program validation. *corpora* represents a program input, and *fuzzing corpus* represents a collection of program inputs.**

- The *library specification* guides LLMs to generate code that adheres to specified patterns required by the libraries. Some library API functions may read inputs from files, file streams, or file descriptors, which may deviate from the standard routines of fuzz drivers. By incorporating the relevant library specifications into the prompts for LLMs, we help LLMs generate code patterns that adhere to these specifications.

Once PromptFuzz fills in these components to generate a prompt, it queries LLMs with the prompt to generate programs.

## 3.3 Erroneous Program Validation

Limited by training data bias and imperfect code synthesis ability of LLMs, the code generated by LLMs may be erroneous [36, 51, 54]. A good target for fuzzing should be, at the very least, free of any errors in the code itself so that all runtime errors are attributed to the library code that the target calls [22, 70]. Since LLMs are unable to generate error-free programs consistently, we developed a technique for identifying erroneous programs generated by LLMs.

Identifying syntactic errors is easy, but identifying semantic errors is tricky. In programs for testing libraries, semantic errors include both library misuses and general bugs, such as memory bugs and incorrect control flows. Precisely identifying these errors is challenging because it requires in-depth knowledge of the library and complicated static and dynamic analyses that are likely slow.

Prior attempts to tackle these challenges mainly focused on leveraging existing knowledge to detect buggy code patterns. Rule-based approaches required manual rules for detection [10, 27, 37]. Learning-based approaches mined correct or incorrect usage from sources, such as consumer code, code patches, or documents [31, 32, 42, 45, 46, 62, 66, 69]. Unfortunately, neither approach was effective. First, they, particularly learning-based methods, suffered from poor precision and recall [2], resulting in either high false positive rates in detecting library bugs or fuzz drivers with low code coverage. Second, gathering learning materials and writing rules were laborious, and the materials and rules could not be shared across libraries.

To address these challenges, PromptFuzz eliminates erroneous programs in three steps (Figure 4):

(1) It removes programs with syntax errors as identified by C/C++ compilers.
(2) It compiles the remaining programs into executables and incorporates multiple runtime sanitizers, which capture and analyze deviations from expected behavioral patterns.

(3) PromptFuzz fuzzes these programs using the provided corpus and removes any program where the sanitizers detect deviations. During fuzzing, PromptFuzz adds the inputs that trigger unique behavior to the corpus, expanding the corpus for more thorough runtime-based validation. After fuzzing, PromptFuzz calculates the code coverage achieved by the programs and removes those that fail to meet the code coverage criteria, indicating the sufficient exercise of the library API functions.

*3.3.1 Fuzzing Validation.* To keep as many valid generated programs as possible for fuzzing libraries, PromptFuzz conservatively removes only programs exhibiting identifiable errors. For this purpose, PromptFuzz compiles the programs with several runtime error sanitizers — Address-Sanitizer (ASan [56]), Undefined-Behavior-Sanitizer (UBSan [41]), and File-Sanitizer (see in Section 4.2) — which can identify violations based on predefined rules with no false positives. PromptFuzz then executes the programs with a provided corpus, monitors their runtime behavior, and removes the programs where the sanitizers report violations. To minimize API misuse in the generated programs, PromptFuzz does not differentiate whether the reported violations occurred in the generated code or library code because the latter could also be triggered by API misuse. We analyzed this rule and found that it removed many generated programs with API misuses even though it occasionally removed generated programs that triggered true library bugs.

Initially, the program's inputs for fuzzing can be either the seed inputs [21] provided by the developers or simply empty. These seed inputs typically consist of representative inputs for certain fuzz drivers and therefore serve as a good starting point for fuzzing. However, as LLMs generate these programs using various API functions to accomplish diverse tasks, as explained in Section 3.2, many programs may require custom inputs, so the initial seed inputs may be unsuitable for those programs. This reduces code coverage and leads to undiscovered errors.

To maximize the range of code PromptFuzz can examine, PromptFuzz continuously evolves the fuzzing corpus throughout the fuzzing process. After a program passes the previous validation processes, PromptFuzz uses a grey-box fuzzer to mutate its inputs while simultaneously monitoring code coverage. If the code coverage increases within each time interval (e.g., 60 seconds), PromptFuzz continues fuzzing until it exhausts the time budget (e.g., 600 seconds). Subsequently, PromptFuzz adds the inputs that have triggered new code coverage into the fuzzing corpus. The objective here is to generate specific inputs required by each program, rather than triggering 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, so this continuous evolution improves the likelihood of generating suitable program inputs over time.

To show how the fuzzing process works, we take the program generated by LLMs shown in Figure 1 as an example. This program uses the libvpx API functions to decode a fragment of encoded video frames and processes each frame iteratively. However, the initial seed inputs of libvpx was a collection of video stream files (i.e., IVF) that contained both headers and video frames, which were unsuitable for the scenario depicted in Figure 1. When PromptFuzz executed the program directly on the initial seed inputs, the call to `vpx_code_decode()` (line 17) returned a `VPX_CODEC_UNSUP_BITSTREAM` error, causing the program to exit immediately at line 20. The sanitizers failed to report the potential risks in the code at lines 23–32 because of the program's early exit. After multiple rounds of mutation of the initial fuzzing corpus, PromptFuzz generated a suitable input for `vpx_codec_decode()` that passed the error checking at line 18, which allowed the sanitizers to validate the hitherto unchecked code after this line.

*3.3.2 Coverage Validation.* Fuzzing validation (Section 3.3.1) accurately identifies errors violating sanitizer rules. However, due to the nature of runtime analysis, it struggles to validate the code that is hard to reach. For example, in line 17 of Figure 1, the code will not execute if the call to `vpx_codec_dec_init()` in line 12 contains incorrect arguments, but the sanitizers are unable to detect this API misuse because it does not violate any sanitization rules. To overcome this limitation, PromptFuzz validates programs based on their runtime code coverage.

To conduct coverage validation, PromptFuzz gathers code coverage for the remaining programs and identifies the *critical paths*, defined as the paths that contain the greatest number of library API calls in the control flow graph of a program. Critical paths represent the API usage that we want to test rather than error-handling code. For example, the critical path in Figure 1 executes lines 11, 12, 17, 25, 27, and 31, capturing the essential API calls within the program. Based on this analysis, PromptFuzz removes the programs where an API call on a critical path has not been executed. This process encourages important API usage to be thoroughly tested. We concentrate on critical paths rather than all program paths, as certain error-handling code is difficult to access or requires specific configurations.

Coverage validation has two benefits. First, prior methods that relied solely on runtime validation could not determine the correctness of unreachable code in programs. Although fuzzing is employed to evolve the fuzzing corpus, it does not guarantee that all programs have been generated with suitable inputs. Coverage validation removes many programs with unreachable API calls. While this approach may mistakenly exclude programs without API misuse, it significantly reduces false positives in bug detection. Second, certain library API uses do not trigger abnormal behavior that runtime sanitizers can capture, e.g., the erroneous API initialization in Figure 1. Coverage validation can exclude the programs containing these API misuses.

## 3.4 Coverage-Guide Prompt Mutation

To create prompts for successive rounds, PromptFuzz mutates the API combinations within the previous prompts. Although LLMs are able to generate programs by combining different API functions, randomly combining them in the prompts would be inefficient. PromptFuzz employs API-level power scheduling and prompt-level mutation strategies to generate effective prompts using code coverage as feedback.

*3.4.1 Power Schedule.* PromptFuzz schedules API functions in LLM prompts based on their energy. More prompts containing a particular API function often correlate with more code coverage but also raise query costs. Beyond a certain point, if the code coverage for an API function plateaus, the benefit of prompting it diminishes. Therefore, an effective power scheduling strategy should maximize library code coverage while minimizing the number of LLM queries. Drawing inspiration from AFLFast[7], PromptFuzz implements monotonous power scheduling, which reduces the energy of well-tested API functions.

Initially, PromptFuzz assigns equal energy for each API function and maintains a set of visited branches and call graphs for the library under test. During each iteration of fuzzing, it updates the visited branches and calculates each API function's branch coverage, as shown in Equation 2. When computing an API function's branch coverage, it considers not only the branches within the function's body but also the branches within the bodies of any recursive callees.

$$\mathrm{cov}(i) = \frac{\text{covered branches inside } i}{\text{total branches inside } i} \tag{2}$$

For an API function $i$, PromptFuzz updates its energy $energy(i)$ following the exponential schedule in AFLFast [7], shown in Equation 3:

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

where $\mathrm{seed}(i)$ is the number of seed programs that call $i$, $\mathrm{prompt}(i)$ is the number of prompts that contain $i$, and $E$ is an exponent to regulate the frequency of $i$. As a result, the fewer times PromptFuzz has exercised an API function, the higher energy PromptFuzz assigns to the function, and the higher probability PromptFuzz will include the function in future prompts with.

#### 3.4.2 *Mutation strategies.*

PromptFuzz mutates API function combinations in prompts to instruct program generation. PromptFuzz borrows the following mutation strategies, which are commonly used in traditional fuzzers [16, 20, 23, 61, 68], but applies them to API functions in prompts:

- Insertion$(C, A)$: Insert API function $A$ into combination $C$.
- Replacement$(C, A, B)$: Replace API function $A$ in combination $C$ with API function $B$.
- Crossover$(C, S)$: Merge combinations $C$ and $S$ into a new combination.

Guided by the energy of API functions, PromptFuzz schedules the mutations to assemble API function combinations to generate previously unexplored functions. However, combining API functions based solely on their degrees of exploration without considering their dependencies prevents LLMs from exploring complex API relations. To overcome this limitation, for each seed program, PromptFuzz gathers the following statistics, which reflect the effectiveness of API combinations in prompts:

- Density: The maximum number of library API calls sharing explicit data dependency.
- Unique branches: The number of unique branches triggered during program execution.

PromptFuzz quantifies a program's quality by Equation 4, which assigns a higher quality to the programs that have more correlated API calls and that discover more branches.

$$\mathrm{quality}(g) = \mathrm{density}(g) \times (1 + \mathrm{unique\_branches}(g)) \tag{4}$$

During each iteration of fuzzing, PromptFuzz explores the seed bank and updates the qualities of those seed programs. Using the feedback from library API energies and seed qualities, PromptFuzz applies Algorithm 1 to select a new API combination to be used in the next iteration. If the current iteration has insufficient seed programs, PromptFuzz enters the warm-up stage (lines 3–7 in Algorithm 1), which randomly selects high-energy API functions to explore previously undiscovered library usage. In the mutation stage (lines 9–23 in Algorithm 1), PromptFuzz uses the sequence of API calls on the seed program's critical path as the pivot for mutation, where it discards the API calls that do not interact with others. Focusing mutation on the pivot allows PromptFuzz to explore intricate API usage. Finally, PromptFuzz uses the new API combination to construct a prompt for the next iteration of program generation.

**Algorithm 1** Selecting a new API combination

```
1:  function MUTATION($APIs, Seeds$)
2:      $Comb = \{\}$
3:      if $WarmUp(Seeds)$ then
4:          while $len(Comb) < DefaultLen$ do
5:              $A \leftarrow ChooseByEnerge(APIs)$
6:              $Insert(Comb, A)$
7:          end while
8:          return $Comb$
9:      end if
10:     $seed \leftarrow ChooseSeedByQuality(Seeds)$
11:     $Comb \leftarrow CriticalCalls(seed)$
12:     $mutator \leftarrow RandChoose(Insert, Replace, Crossover)$
13:     switch $mutator$ do
14:         case $Insert$
15:             $A \leftarrow ChooseByEnerge(APIs)$
16:             $Insert(Comb, A)$
17:         case $Replace$
18:             $A \leftarrow Choose(Comb)$
19:             $B \leftarrow ChooseByEnerge(APIs)$
20:             $Repalce(Comb, A, B)$
21:         case $Crossover$
22:             $seedB \leftarrow ChooseSeedByPrevious(seed, Seeds)$
23:             $CombB \leftarrow CriticalCalls(seedB)$
24:             $Comb = CrossOver(Comb, CombB)$
25:     return $Comb$
26: end function
```

## 3.5 Constrained Fuzzer Scheduling

In the final stage, PromptFuzz combines seed programs into a fuzz driver and schedules it to be fuzzed. We mark the seeds that trigger unique branches as *unique seeds*. Since the *unique seeds* encompass nearly all discovered API functions, only they are included in the fuzz driver.

To empower seed programs for fuzzing, PromptFuzz infers the constraints on API function arguments from the programs stored in the seed bank. Under these constraints, PromptFuzz instruments the *unique seeds* by converting their API function arguments from constants to variables capable of receiving arbitrary bytes from the fuzzers. Finally, PromptFuzz integrates these seeds into a single fuzz driver, which schedules each seed randomly.

*3.5.1 Argument Constraint Inference.* PromptFuzz infers the argument constraints of API functions via statistical inference. For arguments of an immutable array or scalar type, PromptFuzz infers them as potential recipients for random bytes from the fuzzers. These arguments are commonly subject to the following constraints, which can significantly affect the effectiveness of fuzz drivers [13, 25]:

- ArrayLength$(A, n)$: $n$ is the capacity of the array $A$.
- ArrayIndex$(A, i)$: $i$ is an index to the array $A$.
- FileName$(S)$: $S$ is a file path.
- FormatString$(S)$: $S$ is a format string.
- AllocSize$(n)$: $n$ is the size of buffer allocation.
- FileDesc$(fd)$: $fd$ is a file descriptor.

PromptFuzz infers these constraints by the following static analyses on seed programs:

- ArrayLength$(A, n)$: Check for statements that indicate $n$ as the size of $A$, such as `malloc`, `sizeof`, and `strlen`.
- ArrayIndex$(A, i)$: If $i$ is a scalar and if its value is always smaller than the length of $A$.
- FileName$(S)$: If $S$ is assigned the string `input_file` or `output_file` (Figure 3).
- FormatString$(S)$: If $S$ contains the character `%`.
- AllocSize$(n)$: PromptFuzz infers this constraint by varying the arguments of scalar types between their minimum and maximum values, and then executing them with the fuzzing corpus and observing memory allocation sizes. If the sizes differ significantly, PromptFuzz infers this constraint.
- FileDesc$(fd)$: Examine the data flow of the return value of the calls that are related to file descriptors, such as `open` and `fileno`.

If PromptFuzz infers multiple constraints on the same argument, it takes the constraint that has been inferred the most times.

*3.5.2 Constrained Argument Conversion.* PromptFuzz converts the arguments of constant values into variables of the same corresponding type where the variables can accept arbitrary bytes from the fuzzers. PromptFuzz implements a custom FuzzedDataProvider [39] to segment bytes from the fuzzers into multiple sections and convert each section into a value of its designated type. To generate values of variables with statically known sizes, such as scalars and fixed-size arrays, it consumes bytes of the required size and statically casts them. For dynamically sized variables, it consumes bytes until it encounters specific magic bytes. After that, PromptFuzz adjusts the value to satisfy the inferred constraint on the argument. For arguments with the FileName, FormatString, AllocSize, or FileDesc constraints, PromptFuzz retains their original constant values. For arguments with the ArrayLength or ArrayIndex constraints, PromptFuzz ensures that each value is no larger than the corresponding array length. For each converted argument, PromptFuzz attempts to provide them with several different random values. If PromptFuzz's sanitizers detect an error, indicating an erroneous conversion, PromptFuzz cancels the conversion.

*3.5.3 Fuzzer Scheduling.* PromptFuzz consolidates the seed programs into a fuzz driver that schedules each seed program based on several specific bytes provided by the fuzzers. To ensure efficient fuzzing, the fuzzing corpus in Section 3.3.1 serves as the initial input for this fuzz driver. PromptFuzz gathers the constant values of the converted arguments to form their initial corpora.

## 4 Implementation

We implemented PromptFuzz in 17 595 lines of Rust code and have made the source code available in our repository [67]. 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 [58] 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 3.5.3, 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 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 [55] over several years. We compared the code coverage achieved by PromptFuzz's fuzz drivers with other approaches for fuzz driver generation. Secondly, we evaluated the ability of the fuzz drivers generated by PromptFuzz to find bugs. Lastly, we analyzed the key components of PromptFuzz to demonstrate how each component contributes 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 [40] was the grey-box fuzz engine used in all evaluations.

**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** | **Query Cost / Time** | **PromptFuzz** | **OSS-Fuzz** | **Hopper** | **UC** | **VB** | **C** |
| **curl** | 8.4.0 | 154K | 93 | 26 644 | 2550 | 664(106) | \$3.58 / 12h32m | 5283(**19.82%**) | #20 / 822(3.09%) | 3383(12.69%) | 0 | 0 | 0 |
| **libTIFF** | 4.6.0 | 108K | 195 | 14 204 | 2510 | 153(71) | \$4.02 / 11h10m | 7448(**52.43%**) | #1 / 5740(40.60%) | 3932(27.68%) | 6 | 6 | 6 |
| **libjpeg-turbo** | 3.0.1 | 144K | 77 | 10 972 | 2730 | 180(82) | \$4.98 / 22h44m | 5186(47.26%) | #9 / 6187(**56.39%**) | 3971(36.19%) | 4 | 2 | 2 |
| **sqlite3** | 3.43.2 | 413K | 289 | 38 056 | 2210 | 404(74) | \$2.90 / 28h10m | 28 016(**73.61%**) | #1 / 9760(25.64%) | 10 855(28.52%) | 5 | 3 | 3 |
| **libpcap** | 1.10.4 | 58K | 84 | 7816 | 2580 | 151(49) | \$3.68 / 9h30m | 2974(39.25%) | #3 / 3145(41.51%) | 3686(**47.15%**) | 5 | 3 | 3 |
| **cJSON** | 1.7.16 | 10K | 76 | 1020 | 2680 | 209(54) | \$3.40 / 5h1m | 846(82.94%) | #1 / 475(46.57%) | **900(88.23%)** | 5 | 3 | 3 |
| **libaom** | 3.7.0 | 530K | 47 | 61 702 | 2290 | 237(59) | \$4.11 / 16h26m | 15 811(**25.62%**) | #1 / 10 984(18.01%) | 7493(12.14%) | 3 | 3 | 3 |
| **libvpx** | 1.13.1 | 362K | 40 | 35 544 | 3430 | 396(98) | \$6.16 / 14h34m | 7434(**20.91%**) | #2 / 4721(13.32%) | 3603(10.13%) | 4 | 4 | 4 |
| **c-ares** | 1.20.0 | 59K | 61 | 4038 | 1590 | 126(38) | \$2.47 / 8h42m | 2141(53.02%) | #2 / 791(22.80%) | 2932(**72.61%**) | 3 | 2 | 2 |
| **zlib** | 1.3 | 30K | 87 | 2894 | 1630 | 259(82) | \$2.41 / 11h10m | 2210(76.36%) | #9 / 1525(52.80%) | 2284(**78.92%**) | 1 | 0 | 0 |
| **re2** | bc0faab | 28K | 70 | 4940 | 2140 | 101(23) | \$3.30 / 13h2m | 3192(64.61%) | #1 / 3900(**78.94%**) | 3403(68.88%) | 0 | 0 | 0 |
| **lcms** | 2.15 | 45K | 286 | 8806 | 9170 | 402(96) | \$14.04 / 34h42m | 3742(**42.49%**) | #8 / 3049(34.62%) | 2043(23.20%) | 2 | 0 | 0 |
| **libmagic** | FILE5_45 | 33K | 18 | 7440 | 2010 | 217(32) | \$2.41 / 6h53m | 4697(**63.67%**) | #3 / 4628(62.74%) | 4377(58.83%) | 4 | 4 | 1 |
| **libpng** | 1.6.40 | 57K | 246 | 7732 | 3560 | 286(99) | \$5.68 / 9h32m | 3906(**50.51%**) | #1 / 1967(25.44%) | 3847(49.75%) | 2 | 0 | 0 |
| **Total** | - | 2M | 1669 | 231 808 | 41 080 | 3785(963) | \$63.14 / 204h8m | 92 886(**40.07%**) | 57 694(24.88%) | 56 709(24.46%) | **44** | **30** | **27** |

**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.

## 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 chose gpt-3.5-turbo-0613; otherwise, we chose gpt-3.5-turbo-16k-0613, which comes at a higher cost but allows for a longer length limit. We set the temperature parameter at 0.9 for the LLMs and sampled 10 programs per query. The default API combination length was 5, the exponent $E$ in the power schedule was 1, and the initial prompt was constructed using a combination of 5 randomly selected library API functions. In the evaluations of PromptFuzz, the fuzzing loop was operated continuously until 10 consecutive iterations passed without discovering new branches, without any restrictions on time and query budget. Each library's consolidated fuzz driver was executed under a 24-hour timeout. Every experiment was repeated five times to mitigate statistical errors, 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. In total, PromptFuzz successfully **generated** 3785 **seed programs for these 14 libraries within 204 hours with the cost of \$63.14 for querying the LLMs** (\$4.15 per library on average) [1]. Overall, the fuzz drivers generated by **PromptFuzz achieved a branch coverage of 40.07% on the tested libraries**, which was 1.61x greater than OSS-Fuzz and 1.63x greater than Hopper, and detected 30 previously unknown bugs during 24-hour experiments. All bugs found have been reported to the corresponding communities. In the following sections, we will detail the results of our evaluations.

## 5.2 Effectiveness on Code Coverage

To evaluate how effective the fuzz drivers generated by PromptFuzz on code coverage are, we compared 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 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. For Hopper, which assembles fuzz drivers and performs fuzzing on libraries simultaneously, we ran Hopper on each library for a period of 24 hours plus the time PromptFuzz took to generate fuzz drivers for that library, thereby ensuring a fair comparison.

The evaluation results are shown in Table 1. In comparing 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 the libraries totally. 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.24.46%) as well. There are two main reasons for PromptFuzz's better performance compared to Hopper. Firstly, leveraging internal knowledge of LLMs, PromptFuzz effectively extracts complex information about API interdependency in various library API functions 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 API functions, such as *libaom*, *libvpx*, and *re2*, because it lacks support for conditional grammars. In contrast,

[1] At the time of experiments, the input and output prices for gpt-3.5-turbo-0613 were 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 2: Previous known bugs found by PromptFuzz**

| ID | Library | Location | Buggy Function | Bug Type | Status | Track ID |
|---|---|---|---|---|---|---|
| 1. | libaom | aom_dsp/x86/highbd_varianc_sse2.c:49:7 | highbd_8_variance_sse2 | Segment Violation | Confirmed | [3489] |
| 2. | libaom | av1/encoder/ratectrl.c:2501:7 | av1_rc_update_framerate | Uninitialized Stack | Confirmed | [3509] |
| 3. | libaom | av1/encoder/encoder.h:3886:12 | timebase_units_to_ticks | Integer Overflow | Confirmed | [3510] |
| 4. | libvpx | vp8/vp8_dx_iface.c:133:3 | vp8_peek_si_internal | Segment Violation | Confirmed | [1817] |
| 5. | libvpx | vp8/vp8_dx_iface.c:252:47 | update_fragments | Buffer Overflow | Confirmed | [1827] |
| 6. | libvpx | vp8/vp8_cx_iface.c:951:56 | vp8e_encode | Integer Overflow | Confirmed | [1828] |
| 7. | libvpx | vp8/encoder/encodeframe.c:448:18 | encode_mb_row | Integer Overflow | Confirmed | [1831] |
| 8. | libmagic | apprentice.c:3289:11 | apprentice_map | Buffer Overflow | Waiting | [481] |
| 9. | libmagic | magic.c:617:19 | magic_setparam | Buffer Overflow | Waiting | [482] |
| 10. | libmagic | apprentice.c:3358:6 | check_buffer | Buffer Overflow | Confirmed | [483] |
| 11. | libmagic | softmagic.c:1675:11 | mget | Integer Overflow | Waiting | [486] |
| 12. | libTIFF | tif_unix.c:222:12 | TIFFOpen | Out of Memory | Confirmed | [614] |
| 13. | libTIFF | tif_pixarlog.c:776:28 | PixarLogSetupDecode | Out of Memory | Confirmed | [619] |
| 14. | libTIFF | tif_read.c:546:10 | TIFFReadEncodedStrip | Out of Memory | Confirmed | [620] |
| 15. | libTIFF | tif_getimage.c:621:14 | TIFFReadRGBAImageOriented | Out of Memory | Confirmed | [620] |
| 16. | libTIFF | ti_strip.c:333:9 | TIFFRasterScanlineSize64 | Out of Memory | Confirmed | [621] |
| 17. | libTIFF | tif_getimage.c:3345:9 | TIFFReadRGBATileExt | Segment Violation | Confirmed | [622] |
| 28. | sqlite3 | sqlite3.c:178513:23 | sqlite3_unlock_notify | Null Pointer Dereference | Confirmed | [e77a5] |
| 19. | sqlite3 | sqlite3.c:133068:23 | sqlite3_enable_load_extension | Null Pointer Dereference | Confirmed | [9ce83] |
| 20. | sqlite3 | sqlite3.c:174382:23 | sqlite3_db_config | Null Pointer Dereference | Confirmed | [5e3fc] |
| 21. | c-ares | lib/ares_getaddrinfo.c:2173:8 | config_sortlist | Memory Leak | Confirmed | [d62627] |
| 22. | c-ares | lib/ares_getaddrinfo.c:2184:8 | config_sortlist | Memory Leak | Confirmed | [d62627] |
| 23. | libjpeg-turbo | turbojpeg.c:2245:46 | tj3DecodeYUV8 | Integer Overflow | Confirmed | [78eaf0] |
| 24. | libjepg-turbo | turbojpeg-mp.c:366:29 | tj3LoadImage16 | Out of Memory | Confirmed | [735] |
| 25. | libpcap | pcap-linux.c:381:32 | pcap_create | File Descriptor Leak | Confirmed | [1233] |
| 26. | libpcap | pcap-linux.c:354:6 | pcapint_create_interface | Null Pointer Dereference | Confirmed | [1239] |
| 27. | libpcap | pcap-util.c:466:60 | pcapint_fixup_pcap_pkthdr | Misaligned Address | Confirmed | - |
| 28. | cJSON | cJSON.c:394:28 | cJSON_SetNumberHelper | Type Error Cast | Confirmed | [805] |
| 29. | cJSON | cJSON.c:2448:30 | cJSON_CreateNumber | Type Error Cast | Confirmed | [806] |
| 30. | cJSON | cJSON.c:1892:83 | cJSON_DeleteItemFromObjectCaseSensitive | TimeOut | Confirmed | [807] |
| 31*. | libaom | av1/encoder/encoder.c:2605:16 | encode_without_recode | Segment Violation | Confirmed | [3534] |
| 32*. | libvpx | vpx_tpl.c:140:29 | vpx_free_tpl_gop_stats | Segment Violation | Confirmed | [1837] |
| 33*. | curl | urlapi.c:1245:3 | parseurl | Assertion Failure | Confirmed | [12775] |

**Location**: The source file location of the crash point. **Track ID**: The ID used for tracking this issue in their corresponding bug tracker system.
The bugs with IDs 31, 32 and 33 were detected through 15 days of fuzzing.

PromptFuzz can support 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

To demonstrate the effectiveness of PromptFuzz in detecting library bugs, we analyzed the crashes reported by the fuzz drivers generated by PromptFuzz during their fuzzing time. Duplicate crashes were removed by examining the call traces. Throughout the 24-hour fuzzing period, 44 unique crashes were reported. After manually reviewing the code and documentation to verify the validity of these unique crashes, we identified 30 of them as valid bugs. All identified bugs have been reported to the respective communities for confirmation and resolution. At the time of writing, 27 identified bugs have been confirmed, while the remaining three await responses. The details of these bugs are available in Table 2.

*5.3.1 False Positive Analysis.* We analyzed the root causes of the ineffective warnings produced by previous fuzzing. Among the 14 ineffective warnings, there were 8 warnings resulting 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 API calls entering error states and returning null pointers. If the subsequent library API calls access these null pointers without implementing handling for those null pointer arguments, spurious crashes may occur. We must note that we do not consider these crashes false positives in PromptFuzz's bug detection. Instead, they are robustness issues stemming from the library API functions failing to handle the passed null pointers. Excluding the 8 warnings reported due to library API robustness issues, **only 6 crashes were identified as false positives in PromptFuzz's bug detection. We argue that PromptFuzz achieves a detection accuracy of 86.36% (38/44)**. Among these 6 false positives, 2 crashes were caused by constraints that PromptFuzz failed to infer from the libraries. Those constraints haven't been deduced because LLMs failed to generate the correct usage for the corresponding library API functions, hence causing the conversion of their arguments to trigger violations. The remaining 4 positives were considered misuse of the target library that escaped PromptFuzz's validation due to their complex triggering mechanisms. For examples, a false positive found in zlib could only be triggered by special values[2] and an issue found in libpng requires passing a consistent set of arguments to `png_write_png`[3].

*5.3.2 False Negative Analysis.* In our preceding fuzzing experiments, OSS-Fuzz found no bugs, while Hopper identified 5 valid bugs. To investigate the false negative rate of PromptFuzz in detecting library bugs, we selected the 5 bugs identified by Hopper and the 17 bugs presented in Hopper's paper, all of which were

[2]https://github.com/madler/zlib/issues/904
[3]https://github.com/pnggroup/libpng/issues/491

**Table 3: Impact of eliminated erroneous programs and inferred argument constraints**

| Library | Total | Eliminated Erroneous Programs | | | Remain | Inferred Argument Constraints | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Syntax | Fuzzing | Coverage | | ArrayLength | ArrayIndex | Format | FileName | FileDesc | AllocSize |
| curl | 2550 | 1291 | 472 | 123 | 664 | 13 / 13 / 0 | 0 / 0 / 0 | 10 / 5 / 0 | 2 / 2 / 1 | 3 / 3 / 0 | 0 / 0 / 0 |
| libTIFF | 2510 | 1303 | 994 | 60 | 153 | 25 / 19 / 1 | 0 / 0 / 0 | 0 / 0 / 0 | 6 / 3 / 2 | 3 / 1 / 1 | 4 / 4 / 0 |
| libjpeg-turbo | 2730 | 1948 | 267 | 335 | 180 | 25 / 23 / 3 | 0 / 0 / 2 | 0 / 0 / 0 | 8 / 8 / 0 | 0 / 0 / 0 | 12 / 10 / 0 |
| sqlite3 | 2210 | 920 | 638 | 248 | 404 | 25 / 10 / 0 | 0 / 0 / 0 | 7 / 3 / 0 | 12 / 2 / 0 | 0 / 0 / 0 | 4 / 0 / 0 |
| libpcap | 2580 | 1232 | 583 | 614 | 151 | 3 / 3 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 5 / 5 / 1 | 0 / 0 / 2 | 0 / 0 / 0 |
| cJSON | 2680 | 562 | 1630 | 279 | 209 | 7 / 7 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 1 / 1 / 0 |
| libaom | 2290 | 1437 | 244 | 372 | 237 | 7 / 7 / 0 | 1 / 1 / 0 | 0 / 0 / 0 | 0 / 0 / 1 | 0 / 0 / 0 | 0 / 0 / 0 |
| libvpx | 3430 | 1943 | 676 | 415 | 396 | 3 / 3 / 0 | 1 / 1 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 0 / 0 / 0 |
| c-ares | 1590 | 863 | 541 | 60 | 126 | 21 / 20 / 0 | 0 / 0 / 1 | 0 / 0 / 0 | 0 / 0 / 0 | 2 / 2 / 0 | 0 / 0 / 0 |
| zlib | 1.630 | 709 | 477 | 185 | 259 | 26 / 25 / 0 | 0 / 0 / 1 | 2 / 1 / 0 | 2 / 2 / 1 | 1 / 1 / 0 | 2 / 2 / 0 |
| re2 | 2140 | 1182 | 814 | 43 | 101 | 6 / 5 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 0 / 0 / 2 | 0 / 0 / 0 |
| lcms | 9170 | 6627 | 2048 | 93 | 402 | 25 / 23 / 2 | 3 / 3 / 3 | 1 / 1 / 0 | 6 / 6 / 0 | 0 / 0 / 0 | 7 / 4 / 0 |
| libmagic | 2010 | 1295 | 276 | 222 | 217 | 2 / 2 / 0 | 0 / 0 / 0 | 0 / 0 / 0 | 6 / 6 / 0 | 1 / 1 / 0 | 0 / 0 / 0 |
| libpng | 3560 | 2521 | 600 | 153 | 286 | 16 / 7 / 0 | 0 / 0 / 1 | 0 / 0 / 0 | 1 / 1 / 0 | 0 / 0 / 0 | 4 / 4 / 0 |
| **Total** | 41 080 | 23 833 | 10 260 | 3202 | 3785 | **197 / 167 / 5** | **5 / 5 / 8** | **20 / 10 / 0** | **48 / 35 / 6** | **10 / 8 / 5** | **34 / 25 / 0** |

The column of **total** refers to the number of generated programs. The column of **remain** refers to the number of programs that passed our validation. The numbers separated by slashes represent the counts of **the ground truth constraints**, **PromptFuzz correctly inferred constraints**, and **PromptFuzz error-inferred constraints**, respectively.

confirmed by developers, as our evaluation benchmark. Out of the five bugs found by Hopper in our experiments, PromptFuzz detected three bugs[4], but missed two. Our manual inspection of the two missed bugs revealed that PromptFuzz had successfully generated the code containing the associated bug patterns, but one was eliminated by PromptFuzz's fuzzing validation, while the other did not generate the input required to trigger a crash. The 17 bugs presented in Hopper's paper were attached with commit IDs related to their bug reporting, and the details of those bugs can be obtained via their commit messages. For each one of them, we determined that PromptFuzz could detect it if its bug pattern appeared in the fuzz drivers generated by PromptFuzz. Consequently, PromptFuzz was able to detect 15 out of these 17 genuine library bugs. The two false negatives were caused by LLM's failure to generate the related buggy code pattern. For instance, in the lcms with 286 library API functions, the API `cmsStageAllocMatrix` hasn't been generated with related buggy code patterns[5] due to the huge API combination search space. In total, PromptFuzz was able to detect 18 of 22 bugs found by the state-of-the-art library fuzzing approaches.

*5.3.3 Long-term Fuzzing.* To demonstrate the consistent detection of new library bugs with longer fuzzing durations, we conducted an additional 15-day fuzzing session on PromptFuzz's fuzz drivers. Consequently, PromptFuzz reported five more unique crashes, three of which were identified as valid library bugs. These three new bugs were also confirmed by developers, and the details of them can be obtained in Table 2.

## 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 3 presents the detailed analysis results for the eliminated erroneous programs and the inferred argument constraints of PromptFuzz in previous experiments.

*5.4.1 Erroneous Programs Elimination.* The numbers of programs eliminated by each process of PromptFuzz's validation are shown in Table 3. It can be observed that the majority of the erroneous programs (23 833, 63.90%) were eliminated due to syntax errors. Additionally, the *fuzzing* validation process of PromptFuzz identified 10 260 programs (27.51%) that exhibited abnormal runtime behaviors. Furthermore, 3202 programs were eliminated by the *coverage* validation due to insufficient code coverage. Among the 10 260 programs eliminated by PromptFuzz's *fuzzing* validation, we analyzed their crash reports to investigate the factors contributing to the abnormal runtime behaviors. The most prevalent issues detected were segmentation violations (3394, 33.07%) and memory leaks (3003, 29.26%) identified by the sanitizers. Specifically, PromptFuzz's FSan (detailed in Section 4.2) detected 324 programs containing opened file leaks.

To investigate whether the programs were correctly eliminated, we conducted a study in which we randomly selected 10 programs for each library eliminated by the *fuzzing* validation and 10 programs eliminated by the *coverage* validation. We reviewed the code of these programs and conducted careful debugging to determine if they had been properly eliminated. The results revealed that almost all 140 programs eliminated by the *fuzzing* validation contained misuses of library API functions. The only exception is a latent resource leak detected by FSan in *libpcap*[6]. This genuine bug 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 in the most commonly used code pattern in *libpcap* would never have been uncovered. For the 140 programs eliminated by the *coverage* validation, 108 of them were confirmed to have erroneous library usage and were correctly eliminated due to the presence of unreachable library API calls. Among these, 25 were caused by incorrect library initialization, and 40 were caused by the wrong API context, and 43 were due to invalid library API configurations. The remaining 32 programs were mistakenly eliminated because the fuzzer

[4]IDs 17, 19, and 30 as shown in Table 2

[5]https://github.com/mm2/Little-CMS/issues/354

[6]https://github.com/the-tcpdump-group/libpcap/issues/1233

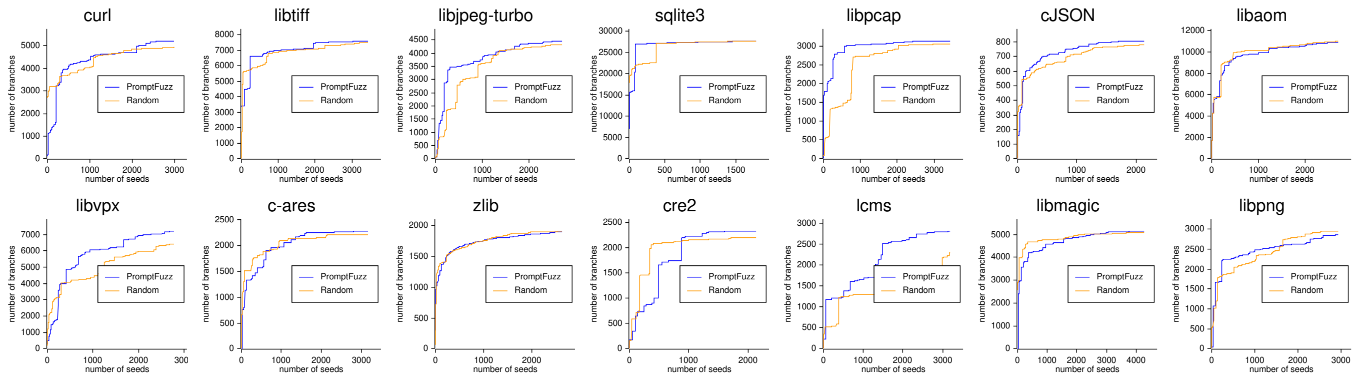


**Figure 5: Branches covered by seed programs generated by PromptFuzz under coverage-guided mutation and blind mutation**

failed to generate input that can reach certain library API calls that are theoretically reachable within the time budget assigned for the *fuzzing* validation process of PromptFuzz (see in Section 3.3.1).

It is important to note that although the validation processes implemented in PromptFuzz may unintentionally exclude correctly functioning and genuine buggy programs, they significantly reduce spurious 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 API functions, 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 3 shows, PromptFuzz achieves 91.24% (250/274) precision and 79.61% (250/314) recall on the inference of argument constraints. The false positives are mainly owing to the absence of argument identifiers in the declarations of library API functions. 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 constants of 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 API functions 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 eliminated 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 API functions to receive random bytes from fuzzers significantly contributed to the discovery of new bugs.

*5.4.3 Coverage Guide Prompt Mutation.* PromptFuzz develops a *coverage-guided mutation* to instruct LLMs in generating valuable programs. To evaluate its effectiveness, we experimented by comparing it with a random blind mutation approach. In this experiment, The blind mutation approach was configured to randomly select library API functions 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 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 API functions, creating programs that reached deeper library states.

The factors leading to the underperformance include the presence of loose coupling between API functions and the large number of API functions within these three libraries. In *libaom*, the API functions exhibit a high degree of coherence, and the interdependency between API functions is evident from their declarations. This clarity facilitates the generation of programs by LLMs, even when provided with randomly selected API functions. For *libpng*, the extensive number of API functions tend to trap the *coverage-guided mutation* setup in local states, while random mutation allows exploration of a broader range of API functions. 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 Discussion & Future Directions

The prototype of PromptFuzz demonstrates the effectiveness of fuzz driver generation for open-source libraries. To give insights into more effective fuzzing approaches, we would like to discuss some essential aspects of PromptFuzz and highlight several possible future directions.

**LLM models choice.** In our experiments, we use GPT3.5 [48] as the LLM model to generate programs using prompts constructed by PromptFuzz. The performance of different models can vary significantly, leading to different outputs and results. More powerful LLM models are prone to bring a more profound understanding of prompted instructions and generate higher-quality programs, indicating that the performance of PromptFuzz may improve if more powerful LLM models are employed.

**Detection accuracy enhancement.** PromptFuzz achieves a high detection accuracy of 86.36%, which is manageable for developers to mitigate library bugs. However, we have identified some strategies that could further enhance its accuracy. Hopper avoids invalid null pointer crashes by inserting assert statements after each library API call that returns a pointer. This approach can help PromptFuzz prevent ineffective warnings caused by null pointer dereferences. Several methods have been proposed for program repair using LLMs [29, 64, 65]. Drawing upon these ideas, we can help PromptFuzz further reduce spurious crashes.

**Application scope extension.** The assessments of PromptFuzz were conducted on open-source libraries. The evaluation results may differ if PromptFuzz is directly applied to closed-source libraries. Nevertheless, fine-tuning LLMs [14, 18] could mitigate these issues. Besides that, we found it is potential to apply PromptFuzz to more types of software, such as web applications or embedded systems. AFGen [38] is a whole-function fuzzing approach that composes fuzz drivers for internal functions of applications. Drawing from this idea, we are also able to generalize PromptFuzz to system or web applications by treating the internal functions as the interfaces.

## 7 Related work

### 7.1 Fuzz Driver Generation.

Several approaches have been proposed to facilitate the generation of fuzz drivers [5, 13, 16, 17, 23–25, 28, 30, 38, 71, 72]. Fudge [5], FuzzGen [24], and UTopia [25] 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 [71] and Winnie [30] 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 [23, 28, 72] have been proposed that generate fuzz drivers without relying on external consumers. GraphFuzz [23] relies on the library specification provided by users to compose fuzz drivers, while RULF [28] relies on strong type restraints in Rust to create fuzz drivers. However, these approaches require human integration or are limited to domain-specific libraries. In addition to the aforementioned approaches, Hopper 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 Hopper. Additionally, TitanFuzz [16] and the method proposed by Google [17] rely on LLMs to generate fuzz drivers, but they struggle to generate fuzz drivers uncover deep library bugs. Compared to these approaches, PromptFuzz generates fuzz drivers without requiring 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 [44] and Goshawk [43] employs natural language processing (NLP) models to identify custom memory management functions within software projects, enhancing static code analysis. CarpetFuzz [60] utilizes NLP to extract API constraints from software documents and detects violations by analyzing the dependencies between API calls. Pythia [4] 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, several approaches are designed directly on pre-trained LLMs [1, 15, 34, 63]. Fuzz4All [63] and *Joshua et.al* fuzz the program parser and language parser by utilizing LLMs to generate and mutate input to the parser. CodaMosa [34] employs LLMs to provide test cases for uncovered functions, addressing coverage plateaus caused by them. GPTFuzz [15] tests the robustness of deep learning library API functions by using LLMs to generate vulnerable cases. Compared with these approaches, 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 high bug 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 explore a wide range of API usage efficiently. We designed oracles for detecting erroneous programs generated by LLMs. By relying on the code synthesis capability of LLMs, PromptFuzz creates fuzz drivers without requiring 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 1.63 times greater than that of Hopper. Additionally, the fuzz drivers generated by PromptFuzz successfully detect 33 new genuine bugs out of 49 crashes, 30 out of which have been confirmed by their communities.

## Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1956364 and by UC Noyce Initiative. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or UC.

## References

[1] Ackerman, J., and Cybenko, G. Large language models for fuzzing parsers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop* (2023), FUZZING 2023.

[2] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering 45*, 12 (2018), 1170–1188.

[3] Amiet, N. Introducing fuzzomatic: Using ai to automatically fuzz rust projects from scratch. https://research.kudelskisecurity.com/2023/12/07/introducing-fuzzomatic-using-ai-to-automatically-fuzz-rust-projects-from-scratch/.

[4] Atlidakis, V., Geambasu, R., Godefroid, P., Polishchuk, M., and Ray, B. Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498* (2020).

[5] Babić, D., Bucur, S., Chen, Y., Ivančić, F., King, T., Kusano, M., Lemieux, C., Szekeres, L., and Wang, W. 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.

[6] Bi, B., Wu, C., Yan, M., Wang, W., Xia, J., and Li, C. 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.

[7] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), p. 1032–1043.

[8] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems* (2020), pp. 1877–1901.

[9] Chang, O. Taking the next step: Oss-fuzz in 2023. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html.

[10] Chen, L., Pei, Y., and Furia, C. A. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), pp. 637–647.

[11] Chen, P., and Chen, H. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (S&P)* (San Francisco, CA, 5 2018).

[12] Chen, P., Liu, J., and Chen, H. Matryoshka: Fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)* (London, UK, 2019).

[13] Chen, P., Xie, Y., Lyu, Y., Wang, Y., and Chen, H. Hopper: Interpretative fuzzing for libraries. In *ACM Conference on Computer and Communications Security (CCS)* (Copenhagen, Denmark, 2023).

[14] Das, S. Fine tune large language model (llm) on a custom dataset. https://dassum.medium.com/fine-tune-large-language-model-llm-on-a-custom-dataset-with-qlora-fb60abdeba07.

[15] Deng, Y., Xia, C., Yang, C., Zhang, S., Yang, S., and Zhang, L. 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).

[16] Deng, Y., Xia, C. S., Peng, H., Yang, C., and Zhang, L. 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.

[17] Dongge Liu, J. M., and Chang, O. Fuzz target generation using llms. https://google.github.io/oss-fuzz/research/llms/target_generation/.

[18] Ferrer, J. An introductory guide to fine-tuning llms. https://www.datacamp.com/tutorial/fine-tuning-large-language-models.

[19] Google. Google's bard. https://bard.google.com/.

[20] Google. Honggfuzz. https://github.com/google/honggfuzz.

[21] Google. How to prepare the seed corpus for oss-fuzz. https://google.github.io/oss-fuzz/getting-started/new-project-guide/#seed-corpus.

[22] Google. What makes a good fuzz target. https://github.com/google/fuzzing/blob/master/docs/good-fuzz-target.md.

[23] Green, H., and Avgerinos, T. Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), pp. 1070–1081.

[24] Ispoglou, K., Austin, D., Mohan, V., and Payer, M. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 2271–2287.

[25] Jeong, B., Jang, J., Yi, H., Moon, J., Kim, J., Jeon, I., Kim, T., Shim, W., and Hwang, Y. H. Utopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE Computer Society, pp. 746–762.

[26] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. Survey of hallucination in natural language generation. *ACM Computing Surveys 55*, 12 (2023), 1–38.

[27] Jiang, J., Xiong, Y., Zhang, H., Gao, Q., and Chen, X. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2018), ISSTA 2018, Association for Computing Machinery, p. 298–309.

[28] Jiang, J., Xu, H., and Zhou, Y. Rulf: Rust library fuzzing via api dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 581–592.

[29] Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., and Svyatkovskiy, A. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263* (2023).

[30] Jung, J., Tong, S., Hu, H., Lim, J., Jin, Y., and Kim, T. Winnie: Fuzzing windows applications with harness synthesis and fast cloning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)* (2021).

[31] Kang, H. J., and Lo, D. Active learning of discriminative subgraph patterns for api misuse detection. *IEEE Transactions on Software Engineering 48*, 8 (2021), 2761–2783.

[32] Kechagia, M., Devroey, X., Panichella, A., Gousios, G., and van Deursen, A. 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.

[33] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35 (NIPS 2022)* (2022).

[34] Lemieux, C., Inala, J. P., Lahiri, S. K., and Sen, S. 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.

[35] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems* (2020), vol. 33, pp. 9459–9474.

[36] Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[37] Liu, K., Koyuncu, A., Kim, D., and Bissyandè, T. F. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), pp. 1–12.

[38] Liu, Y., Wang, Y., Bao, T., Jia, X., Zhang, Z., and Su, P. Afgen: Whole-function fuzzing for applications and libraries. In *2024 IEEE Symposium on Security and Privacy (SP)* (2024), pp. 11–11.

[39] LLVM. Fuzzeddataprovider. https://github.com/llvm/llvm-project/blob/main/compiler-rt/include/fuzzer/FuzzedDataProvider.h.

[40] LLVM. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[41] LLVM. Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[42] Lv, T., Li, R., Yang, Y., Chen, K., Liao, X., Wang, X., Hu, P., and Xing, L. 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.

[43] Lyu, Y., Fang, Y., Zhang, Y., Sun, Q., Ma, S., Bertino, E., Lu, K., and Li, J. Goshawk: Hunting memory corruptions via structure-aware and object-centric memory operation synopsis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)* (Los Alamitos, CA, USA, may 2022), IEEE Computer Society, pp. 1566–1566.

[44] Lyu, Y., Gao, W., Ma, S., Sun, Q., and Li, J. 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, 2021), Springer-Verlag, p. 129–148.

[45] Nielebock, S., Blockhaus, P., Krüger, J., and Ortmeier, F. Automated change rule inference for distance-based api misuse detection. *arXiv preprint arXiv:2207.06665* (2022).

[46] Nielebock, S., Heumüller, R., Krüger, J., and Ortmeier, F. 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.

[47] OPENAI. How to use chat-based language models. https://platform.openai.com/docs/guides/chat/introduction.

[48] OPENAI. Introducing chatgpt. https://openai.com/blog/chatgpt.

[49] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[50] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems* (2022), pp. 27730–27744.

[51] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 754–768.

[52] Peng, B., Galley, M., He, P., Cheng, H., Xie, Y., Hu, Y., Huang, Q., Liden,

L., Yu, Z., Chen, W., and Gao, J. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813* (2023).

[53] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog 1*, 8 (2019), 9.

[54] Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., and Dolan-Gavitt, B. 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).

[55] Serebryany, K. OSS-Fuzz-google's continuous fuzzing service for open source software. In *Proceedings of the 26th USENIX Conference on Security Symposium (technical sessions)* (2017), USENIX Association.

[56] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (2012), USENIX ATC'12, USENIX Association, p. 28.

[57] Sun, Y., Wang, S., Feng, S., et al. Ernie 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation. *arXiv preprint arXiv:2107.02137* (2021).

[58] Tolnay, D. Clang ast deserializer in rust. https://github.com/dtolnay/clang-ast.

[59] Touvron, H., Martin, L., Stone, K., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[60] Wang, D., Li, Y., Zhang, Z., and Chen, K. Carpetfuzz: Automatic program option constraint extraction from documentation for fuzzing. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA, 2023), USENIX Association.

[61] Wang, J., Chen, B., Wei, L., and Liu, Y. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019).

[62] Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S.-C., and Su, Z. Exposing library api misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), pp. 866–877.

[63] Xia, C. S., Paltenghi, M., Tian, J. L., Pradel, M., and Zhang, L. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[64] Xia, C. S., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery* (2023).

[65] Xia, C. S., and Zhang, L. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[66] Yang, J., Ren, J., and Wu, W. Api misuse detection method based on transformer. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)* (2022), IEEE, pp. 958–969.

[67] Yunlong Lyu, Yuxuan Xie, P. C., and Chen, H. Promptfuzz. https://github.com/PromptFuzz/PromptFuzz.

[68] Zalewski, M. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[69] Zeng, H., Chen, J., Shen, B., and Zhong, H. Mining api constraints from library and client to detect api misuses. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)* (2021), IEEE, pp. 161–170.

[70] Zhang, C., Bai, M., Zheng, Y., Li, Y., Xie, X., Li, Y., Ma, W., Sun, L., and Liu, Y. Understanding large language model based fuzz driver generation. *arXiv preprint arXiv:2307.12469* (2023).

[71] Zhang, C., Lin, X., Li, Y., Xue, Y., Xie, J., Chen, H., Ying, X., Wang, J., and Liu, Y. APICraft: Fuzz driver generation for closed-source SDK libraries. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2811–2828.

[72] Zhang, M., Liu, J., Ma, F., Zhang, H., and Jiang, Y. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), IEEE, pp. 318–327.

[73] Zhao, J., Rong, Y., Guo, Y., He, Y., and Chen, H. Understanding programs by exploiting (fuzzing) test cases. In *Findings of the Association for Computational Linguistics (ACL)* (Toronto, Canada, 2023).

## A Example of converted fuzz driver.

Figure 6 shows a case of constrained argument conversion described in Section 3.5.2.

## B Examples of Library Specification.

As stated in Section 3.2, we can provide the library specification to LLMs if the library requires specialized guidelines. The provided library specifications are filled in our prompt template (Figure 3) and used to facilitate the specific code pattern generation. Among the 14 tested libraries in our experiments, we manually crafted the library specifications for 3 libraries: libmagic, sqlite3, and libTIFF.

```c
#include "FuzzedDataProvider.h"
#include <cJSON.h>

extern "C" int LLVMFuzzerTestOneInput_14(const uint8_t* f_data, size_t f_size) {
//fuzzer vars shim {
    FuzzedDataProvider fdp(f_data, f_size);
    FDPConsumeRawBytesWithNullTerm(const uint8_t *, data, size, fdp)
    FDPConsumeRandomLengthString(char, fuzz_str_1, fuzz_str_sz_1, fdp);
    FDPConsumeFloatingPoint(double, fuzz_var_2, fdp)
    FDPConsumeFloatingArray(float, fuzz_array_3, fuzz_array_size_3, fdp);
//fuzzer shim end}

    // Parse the JSON data
    cJSON *root = cJSON_ParseWithLength(data, size);

    // Add a number value to the root object
- cJSON_AddNumberToObject(root, "pi", 3.14);
+ cJSON_AddNumberToObject(root, fuzz_str_1, fuzz_var_2);

    // Create a float array and add it to the root object
- cJSON *array = cJSON_CreateFloatArray("1.23f, 4.56f, 7.89f", 3);
+cJSON *array = cJSON_CreateFloatArray(fuzz_array_3, fuzz_array_size_3);
    ...
    // Delete the cJSON object
    cJSON_Delete(root);
    return 0;
}
```

**Figure 6: An example of constrained argument conversion.**

```c
// Generate the fuzz driver with the beginning code:
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    magic_t magic = magic_open(MAGIC_NONE);
    if (magic == NULL) {
        return -1;
    }
    // The magic file name is "magic"
    if (magic_load(magic, "magic") == -1) {
        magic_close(magic);
        return -1;
    }
  // complete here
```

**Figure 7: The library specification used for libmagic.**

The remaining 11 libraries were not crafted with additional library specifications.

Figure 7 shows the library specification for libmagic. The library libmagic requires a pre-loaded magic database as an argument to calling the libmagic API functions. To initialize this, the file

```
// Generate the fuzz driver with the beginning code:
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data
    , size_t size) {
  sqlite3* db;
  int rc = sqlite3_open(":memory:", &db);
  if (rc != SQLITE_OK) {
    sqlite3_close(db);
    return 0;
  }
  // complete here
```

**Figure 8: The library specification used for sqlite3.**

```
// Generate the fuzz driver with the beginning code:
extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data
    , size_t size) {
  // write data into input_file.
  FILE *in_file = fopen("input_file", "wb");
  if (in_file == NULL) {return 0;}
  fwrite(data, sizeof(uint8_t), size, in_file);
  fclose(in_file);
  // open input tiff in memory
  std::istringstream s(std::string(data, data + size));
  TIFF *in_tif = TIFFStreamOpen("MemTIFF", &s);
  if (!in_tif)
  {
    return 0;
  }
  // complete here
```

**Figure 9: The library specification used for libTIFF.**

location of the magic database file needs to be passed to the function `magic_load()`. To ensure that LLMs can successfully load the magic database, we explicitly specified the file name of the magic database and prepared it in the correct location in advance.

The library sqlite3 requires actively calling `sqlite3_close()` to close the database opened by `sqlite3_open()`, regardless of whether the database is opened successfully. This code pattern against the common pattern of `*_open()` and `*_close()` pairs. Memory leaks will occur if the API `sqlite3_close()` has not been called to close the opened database. We crafted the library specification in Figure 8 to mitigate the memory leak issues.

The library libTIFF provides multiple methods for opening TIFF (Tag Image File Format) files. For instance, `TIFFOpen()` opens a TIFF file using a file location, `TIFFFdOpen()` opens a TIFF file using a file descriptor, and `TIFFStreamOpen()` opens a TIFF file using a byte stream. In our experiments, LLMs tend to utilize `fmemopen()` to obtain a file descriptor and then use `TIFFFdOpen()` to open the TIFF file. However, if the file descriptor is obtained through `fmemopen()`, `TIFFFdOpen()` will consistently fail. We designed the library specification presented in Figure 9 to ensure the correct code pattern is generated.