Directed Greybox Fuzzing via Large Language Model

Hanxiang Xu
Huazhong University of Science and
Technology
China
xuhx@hust.edu.cn

Yanjie Zhao
Huazhong University of Science and
Technology
China
yanjie_zhao@hust.edu.cn

Haoyu Wang
Huazhong University of Science and
Technology
China
haoyuwang@hust.edu.cn

Abstract

Directed greybox fuzzing (DGF) focuses on efficiently reaching specific program locations or triggering particular behaviors, making it essential for tasks like vulnerability detection and crash reproduction. However, existing methods often suffer from path explosion and randomness in input mutation, leading to inefficiencies in exploring and exploiting target paths. In this paper, we propose HGFuzzer, an automatic framework that leverages the large language model (LLM) to address these challenges. HGFuzzer transforms path constraint problems into targeted code generation tasks, systematically generating test harnesses and reachable inputs to reduce unnecessary exploration paths significantly. Additionally, we implement custom mutators designed specifically for target functions, minimizing randomness and improving the precision of directed fuzzing. We evaluated HGFuzzer on 20 real-world vulnerabilities, successfully triggering 17, including 11 within the first minute, achieving a speedup of at least 24.8× compared to state-of-the-art directed fuzzers. Furthermore, HGFuzzer discovered 9 previously unknown vulnerabilities, all of which were assigned CVE IDs, demonstrating the effectiveness of our approach in identifying real-world vulnerabilities.

CCS Concepts

• **Security and privacy → Software security engineering**.

Keywords

Fuzzing; Directed Greybox Fuzzing; Large Language Model; Opensource Library; Vulnerability

1 Introduction

Fuzzing is a widely adopted and highly effective technique for identifying software vulnerabilities by generating and executing numerous test cases to detect abnormal program behaviors [11, 34].

Over the years, fuzzing has evolved from simple random testing to more sophisticated and targeted strategies. Among these evolutionary paths, **greybox fuzzing** has emerged as a prominent approach that balances efficiency and scalability [41] by combining elements of both blackbox testing with minimal program visibility and whitebox testing with comprehensive program analysis.

Most greybox fuzzers are traditionally coverage-guided, aiming to maximize overall program exploration. As the field advanced, researchers developed **directed greybox fuzzing (DGF)**, which focuses on specific target locations, such as bug-prone regions, making it particularly useful for scenarios like patch testing and crash reproduction [10, 13]. Despite its targeted approach, DGF faces two key challenges that significantly hinder its efficiency: **the complexity of exploration and the randomness of exploitation**. The exploration phase often encounters the issue of path explosion, where numerous infeasible paths are unnecessarily explored, consuming excessive resources. In the exploitation phase, the inherent randomness of input generation and mutation frequently results in irrelevant inputs being tested and fails to satisfy the precise constraints required to trigger vulnerabilities [66].

To improve the efficiency of directed fuzzing, various solutions have been proposed [22, 23, 53]. One approach focuses on optimizing guidance toward target locations by analyzing program characteristics, such as identifying deviation basic blocks [16, 21] or satisfying execution path constraints [30, 62]. Another approach aims to generate inputs with a higher likelihood of reaching target locations, for example, by predicting reachable inputs [67] or instrumenting only the relevant parts of the code [39]. However, existing methods that leverage static analysis and symbolic execution to evaluate path reachability and constrain execution [10, 62] introduce additional computational overhead and struggle with scalability. This inefficiency delays the discovery of vulnerabilities and limits testing effectiveness, **highlighting the need for more intelligent and targeted strategies in both exploration and exploitation phases**.

Recent advances in artificial intelligence, particularly large language models (LLMs), offer new opportunities to address these challenges in DGF. LLMs have demonstrated remarkable capabilities in various domains, including natural language understanding, code generation, and program analysis [26]. Their ability to understand code semantics, infer program logic, and generate contextually relevant code makes them particularly promising for tackling the core difficulties in DGF. Several recent works have begun to leverage LLMs to address key challenges in fuzzing [19, 55], such as driver synthesis [40, 54], input generation [42, 48], and bug detection [31]. However, **the application of LLMs specifically to the dual challenges of exploration complexity and exploitation randomness in DGF remains largely unexplored**.

Building on these insights, we propose a novel framework, i.e., HGFuzzer, that leverages the reasoning and code generation capabilities of LLMs to improve the efficiency of DGF. HGFuzzer tackles the complexity of exploration by transforming the path constraint problem into a code generation task. Specifically, it employs static analysis to identify potential call chains of the target function and guides LLM to analyze an available call chain, infer execution conditions, and generate target harness. The harness constrains execution paths to focus on relevant code regions, thereby reducing unnecessary exploration and computational overhead. For the exploitation phase, HGFuzzer reduces randomness by systematically guiding input generation and mutation. It utilizes LLM to generate reachable seeds that satisfy the execution constraints derived from the call chain analysis. Additionally, HGFuzzer constructs target-specific mutators by analyzing the characteristics of the target function and the associated vulnerability. These custom mutators are tailored to efficiently trigger the target vulnerability by prioritizing input mutations that align with the conditions required to exploit the vulnerability. By integrating these components, HGFuzzer provides an automatic approach to improve the efficiency of DGF.

We evaluate HGFuzzer across a benchmark of 20 real-world vulnerabilities, comparing it against state-of-the-art directed greybox fuzzers, including AFLGo [10], Beacon [21], and SelectFuzz [39]. HGFuzzer demonstrates superior performance by successfully triggering 17 out of 20 vulnerabilities, compared to 5 by AFLGo and SelectFuzz and 6 by Beacon, with 11 vulnerabilities triggered within the first minute of fuzzing. For successfully triggered known vulnerabilities, it achieves at least a 24.8× speedup to baselines. HGFuzzer also achieved a 27.5% improvement in target function hit rate over the best-performing baseline, demonstrating its ability to effectively guide execution paths toward target functions. Additionally, HGFuzzer identified 9 previously unknown vulnerabilities in two open-source libraries, all of which were assigned CVE IDs.

In summary, this paper makes the following contributions:

- We propose a novel framework, HGFuzzer, that integrates LLM to improve the efficiency of directed greybox fuzzing.
- We transform path constraint analysis into a code generation task, introducing LLM-guided reachable seed generation to reduce exploration complexity and a target-specific mutator to mitigate exploitation randomness during fuzzing.
- We implement HGFuzzer and demonstrate its superiority over state-of-the-art directed fuzzers on real-world benchmarks, successfully discovering 9 previously unknown vulnerabilities with CVE IDs.

2 Background

2.1 Directed Greybox Fuzzing (DGF)

DGF is a targeted software testing approach that focuses on efficiently reaching specific program locations or exposing certain program behaviors, such as vulnerable code regions or critical execution paths [51]. Unlike traditional coverage-guided fuzzing, which maximizes coverage indiscriminately, DGF prioritizes inputs closer to pre-defined targets, saving resources and improving efficiency. First introduced by Böhme et al. with AFLGo [10], operating in two phases: **exploration** and **exploitation**. During exploration, the fuzzer aims to uncover as many paths as possible, favoring seeds

that trigger new paths to maximize the potential of reaching targets. Once sufficient paths are uncovered, the exploitation phase focuses on inputs closer to the targets, assigning them more energy to generate mutations that fulfill the testing goals. Recent advancements, such as SemFuzz [59], ParmeSan [45] and FuzzGuard [67], have employed techniques like natural language processing, sanitizer-based annotations and machine learning to automate target identification, further expanding DGF's applicability to scenarios like patch testing, crash reproduction, and detecting complex vulnerabilities.

2.2 LLM for Fuzzing

Recent research has explored the application of LLMs in fuzzing, demonstrating their potential to enhance input generation, driver synthesis, and bug detection processes [25, 27]. Unlike traditional fuzzing methods, which often rely on randomized or rule-based approaches, LLMs utilize their generative capabilities to produce diverse and contextually valid inputs. Tools such as TitanFuzz [14] and FuzzGPT [15] leverage LLMs to improve seed mutation and input diversity, enabling more effective exploration of complex software behaviors. LLMs have also been applied to automate fuzz driver synthesis, a critical step in fuzzing workflows. For instance, Zhang et al. [60] demonstrate the effectiveness of GPT-4 in generating fuzz drivers for library APIs, significantly reducing manual effort. Similarly, InputBlaster [38] and ChatAFL [43] enhance fuzzing for mobile apps and network protocols, achieving higher bug detection rates and improved input validity. Despite these advances, the potential of LLMs to specifically address the challenges of exploration complexity and exploitation randomness in DGF has yet to be thoroughly investigated.

2.3 Challenges and Motivations

Complexity of exploration. Existing approaches in the exploration phase aim to uncover as many execution paths as possible, as new paths increase the likelihood of reaching the target [10, 12, 16]. This is particularly necessary when the initial seeds are far from the target. Some methods also leverage lightweight static analysis to evaluate the reachability of execution paths to the target and use symbolic execution to constrain path exploration [21, 39, 62]. However, these methods face the challenge of path explosion caused by exploring numerous infeasible paths that cannot reach the given target in a library. Furthermore, existing methods fail to directly utilize the semantic results of static analysis. Instead, they convert these results into constraints for symbolic execution, which introduces additional computation overhead.

To address this issue, our core idea is to leverage the semantic understanding and code generation capabilities of LLM. We transform the complex path constraint problem into a code generation task. Specifically, **we collect the call chains of the target function within the target library, use LLM to analyze these call chains, and generate an executable harness that explicitly constrains the execution path to the target**. This approach avoids complex path exploration. As shown in Figure 1, CVE-2017-2897 is an out-of-bounds write vulnerability in libxls 1.4.0. The entry program contains numerous complex execution paths. Traditional directed fuzzers require extensive path exploration before reaching the target, but most of these paths are irrelevant to the
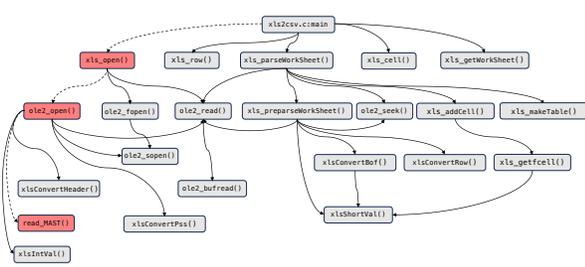
Figure 1: A motivating example from CVE-2017-2897, where the red blocks indicate a reachable path to the vulnerable target function `read_MAST`.

vulnerable function `read_MAST`. By analyzing the call relations of `read_MAST` in the library, we can generate a harness program to constrain the execution path explicitly, significantly simplifying the path exploration process.

```
001 unsigned char *tjLoadImage(const char *filename, ...)
002 {
003   if ((file = fopen(filename, "rb")) == NULL)
004     ...
005   if ((tempc = getc(file)) < 0 || ungetc(tempc, file) == EOF)
006     ...
007   else if (tempc == EOF)
008     ...
009   if (tempc == 'B') {
010     ...
011   } else if (tempc == 'P') {
012     if ((src = jinit_read_ppm(cinfo)) == NULL)
013
014       /* The first character of the input file should be 'P' */
015
016     ...
017   } else
018     ...
019
020   }
021
022 cjpeg_source_ptr jinit_read_ppm()
023 {
024   ...
025   source->pub.start_input = start_input_ppm;
026   ...
027 }
028
029 void start_input_ppm(j_compress_ptr cinfo,...)
030 {
031   ...
032   c = getc(source->pub.input_file);
033   maxval = read_pbm_integer(cinfo, source->pub.input_file, 65535);
034   ...
035   switch (c) {
036   ...
037   case '6':  /* The second character of the input file must be '6' */
038     ...
039     if (maxval > 255) {
040       ...
041     } else if (maxval == MAXJSAMPLE && sizeof(JSAMPLE) == sizeof(U_CHAR) &&
042                (cinfo->in_color_space == JCS_EXT_RGB
043                )) {
044       ...
045     } else {
046
047       /* The maxval of the input file must be less than 255 */
048
050       if (IsExtRGB(cinfo->in_color_space))
051
052       /* The color space of the input file must be extended RGB */
053
054         source->pub.get_pixel_rows = get_rgb_row;
055
056       /* Trigger the vulnerable function */
057
058       else if (cinfo->in_color_space == JCS_CMYK)
059         ...
060       else
061         ...
062     }
063     break;
064   }
065   ...
066 }
```

Figure 2: A motivating example from CVE-2020-13790.

Randomness of exploitation. Fuzzing is inherently a process driven by randomness, where the generation and mutation of inputs often rely on probabilistic techniques [29, 66]. This randomness conflicts with the goal of directed fuzzing, as it can reduce efficiency by diverting resources to inputs that are less likely to reach or trigger the target during the exploitation phase. As shown in Figure 2, CVE-2020-13790 requires satisfying multiple path constraints to execute the vulnerable function `get_rgb_row`. For instance, the input file must start with specific characters ('P' followed by '6'), have a `maxval` less than 255, and use the `JCS_EXT_RGB` color space. Traditional directed fuzzers rely on input without confirmation of target reachability and random mutations to satisfy these constraints, which often results in numerous irrelevant inputs being tested, increasing the time required to reach or trigger the vulnerability. This randomness not only reduces efficiency but also leads to excessive resource consumption, as mutations are not explicitly guided toward satisfying the constraints necessary for exploitation. Addressing this challenge requires reducing randomness in the exploitation phase to better align with the goals of directed fuzzing.

To address this challenge, inspired by existing studies [28, 56, 64], our basic idea is to utilize LLM to reduce the randomness in the exploitation process. Specifically, we first leverage LLM to analyze the complex parameter or variable constraints imposed by the target call chain, enabling the generation of initial reachable inputs for directed fuzzing. Subsequently, we use LLM to further analyze the characteristics of the target function, combined with the potential vulnerability risks, to generate custom mutators tailored for directed fuzzing. This approach aims to minimize the impact of randomness by systematically guiding the fuzzing process.

3 Methodology

Our approach aims to improve the efficiency of directed fuzzing by leveraging the reasoning and code generation capabilities of LLM. As shown in Figure 3, HGFuzzer consists of multiple phases to systematically guide the fuzzing process toward specific target functions. First, it uses static analysis to identify all potential call chains of the target function and filters out the feasible ones (Section 3.1). Then, LLM analyze these call chains to infer the execution conditions required to trigger the target function (Section 3.2). Based on this information, HGFuzzer generates executable target harness and initial input seeds to guide execution paths toward the target (Section 3.3, Section 3.4). Finally, HGFuzzer constructs target-specific mutators using target execution reports to optimize input mutation, reducing the time required to reach the target function and enhancing the effectiveness of the fuzzing process (Section 3.5).

3.1 Call Chain Analysis

Call chain analysis is the initial step of HGFuzzer, where we utilize static analysis to query all existing call chains of the target function within the library and extract an available call chain for further processing. This step provides crucial insights into the invocation context of the target function and forms the basis for subsequent reasoning and harness generation.

A call chain C is represented as a sequence of functions $(F_n, F_{n-1}, \ldots, F_1, F_0)$, where F_0 is the target function and F_n is the starting function of the chain. Let S denote the set of all call chains for the target function. To identify an available call chain, HGFuzzer parses S following the criteria defined in Algorithm 1. If there exists a call
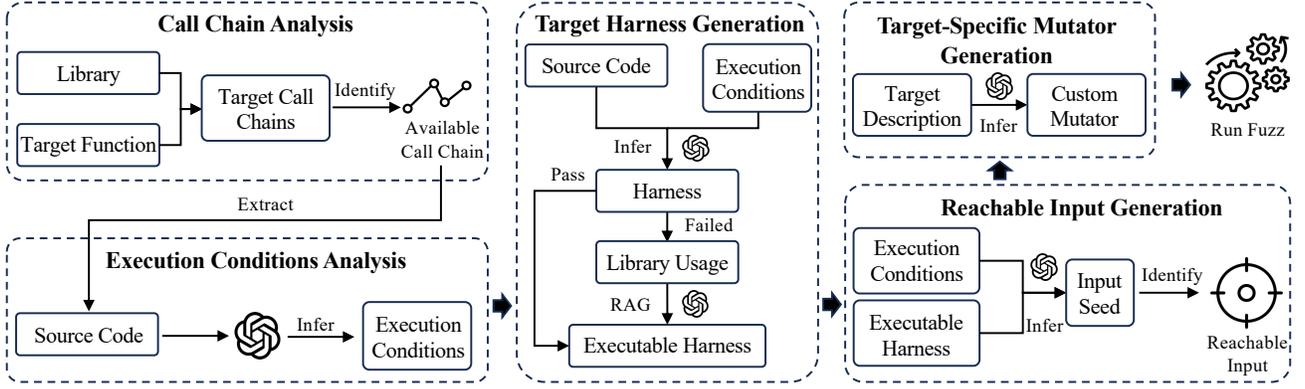
Figure 3: Overview of HGFuzzer.

chain $C \in S$ with $F_n = $ main, HGFuzzer identifies the chain with the smallest length $|C|$ as the available call chain. If no such chain exists, HGFuzzer searches S for the call chain with the smallest length $|C|$ whose starting function F_n is declared as an external function in the library's header files. If such a chain is found, it is identified as the available call chain.

Algorithm 1 Call Chain Identification

Require: Target Function F_0, Library L
Ensure: Available Call Chain C
  1: $S \leftarrow$ CallChains(F_0, L)
  2: $C \leftarrow \emptyset, l \leftarrow \infty$ // l means shortest length of available call chain
  3: **for** $C' \in S$ **do**
  4:    **if** $F_n(C') = $ main $\wedge |C'| < l$ **then**
  5:      $C \leftarrow C', l \leftarrow |C'|$
  6:    **end if**
  7: **end for**
  8: **if** $C = \emptyset$ **then**
  9:    **for** $C' \in S$ **do**
 10:      **if** $F_n(C') \in L$.headers $\wedge F_n(C')$ is extern $\wedge |C'| < l$ **then**
 11:        $C \leftarrow C', l \leftarrow |C'|$
 12:      **end if**
 13:    **end for**
 14: **end if**
 15: **return** C

The preference for call chains starting with main is based on the observation that main and its associated file often provide complete parameter initialization, global variable declarations, and program constraints for calling entry function F_{n-1}. As the primary entry point, main is typically designed for specific functionalities, making it an ideal candidate template example for generating a target harness. When no call chain starting with main can be found, we prioritize call chains beginning with declared external functions, which typically have well-defined interfaces and parameter specifications. External functions, designed as entry points for libraries or modules, generally encapsulate the necessary contextual information

and environment setup for correctly invoking target functionality. This strategy ensures we can identify call chains with sufficient context to build executable target harnesses even without main. In summary, by prioritizing either main-originated chains or those starting with external functions, HGFuzzer ensures the selected call chain provides sufficient initialization context to generate valid and executable target harnesses.

3.2 Execution Conditions Analysis

Given an available call chain $C = (F_n, F_{n-1}, \ldots, F_1, F_0)$, HGFuzzer then extracts the execution conditions required to traverse the chain and ultimately reach the target function F_0. This process involves parsing the source code of all functions in the call chain and using LLM to analyze the conditions for each function call in the chain.

HGFuzzer begins by utilizing static analysis to parse the source code of all functions in C. This parsing step generates a structured representation of the code, enabling precise identification of function definitions and call sites. For each pair of functions <F_i, F_{i-1}> in the chain, where F_i calls F_{i-1}, HGFuzzer employs LLM to analyze the specific execution conditions required for the call. The LLM is designed to complete the following steps:

(1) **Determining the Call Location**: Identifying the exact location in the source code where F_{i-1} is called within F_i. This includes extracting the line number and the surrounding code snippet containing the call.

(2) **Identifying Decision Variables**: Identifying the variables that determine whether the call from F_i to F_{i-1} occurs. These decision variables may include function parameters, global variables, or local variables within F_i.

(3) **Analyzing Conditions**: Analyzing the conditions that these decision variables must satisfy for the call to occur. These conditions are represented as logical predicates or constraints on the variable values, such as inequalities, equality constraints, or ranges.

Figure 4 illustrates the structured output of the LLM for a single function call <F_i, F_{i-1}>. Each decision variable is associated with its constraints, which are expressed in formal logic for clarity and

```
Response of LLM

{
    "current_function":"{current_func}",
    "next_function":" {next_func}",
    "call_location": {
            "line":<line_number>,
            "code_snippet":<exact_code_snippet>
    },
    "decision_variables": [
        {
            "variable": "<variable_name>",
            "conditions": [
                        "<condition_1>",
                        "<condition_2>",
                        "..."
            ]
        },
        "..."
    ]
}
```

Figure 4: An example of the response from the LLM in execution conditions analysis.

precision. By iterating over all function pairs in the call chain, HG-Fuzzer builds a complete representation of the execution conditions needed to reach the target function F_0. This structured analysis accurately captures the dependencies and constraints of the call chain, providing essential information for generating executable harness and deriving inputs that can traverse the chain and trigger the target function.

Illustrative Case. For the motivating example in Figure 2, the call chain C = (tjLoadImage, jinit_read_ppm, start_input_ppm, get_rgb_row) leads to the target function get_rgb_row. The call from tjLoadImage to jinit_read_ppm occurs at line 12 and is conditional on the first character of the input file (temp_c) being 'P' (line 11). Within start_input_ppm, additional conditions must be satisfied, such as the second character of the file being '6' (line 37), the maximum pixel value (maxval) being less than 255 (line 39-45), and the color space of the input file being extended RGB (JCS_EXT_RGB) (line 50). HGFuzzer identifies these decision variables and constraints for each function pair in the call chain, ensuring the conditions required to traverse the chain and execute the target function get_rgb_row are captured in a structured format for downstream use.

3.3 Target Harness Generation

To direct the fuzzing process toward the target function, HGFuzzer generates a targeted fuzz harness based on the available call chain, execution conditions, and corresponding function source code. This harness constrains the exploration space of the fuzzer, reducing redundant path exploration and ensuring efficient reachability of the target function.

Figure 5 presents the prompt template used in HGFuzzer for target harness generation. In addition to the available call chain, function source code, and execution conditions, the input context

```
Prompt Template

**Task:** Generate a targeted C/C++ fuzz harness for a specific fuzzing target within a given library.

**Input Context:**
● **Target Description**: Information about the fuzzing target ({target_info}).
● **Target Function**: The target function to be tested ({target_func}).
● **Available Call Chain**: The complete call chain leading to the target function ({call_chain}).
● **Execution Conditions**: Execution conditions and dependencies for each function in the call chain ({execution_conditions}).
● **Entry Function**: The entry function that initiates the call chain {entry_func_code}.
● **Source File**: Source file where the entry function is defined or called ({src_file}).

**Instructions:**
● Implement a **main** function that calls the entry function {entry_func_name} and guides the execution path along the call chain to the target function.
● Use **argv[1]** as the input file path, compatible with **AFL++ input (@@)**.
● Do not hardcode variables or implement library functions that are already part of the call chain.
● Ensure proper resource management and include necessary headers.
● Design the harness to trigger the target in the shortest possible fuzzing time.
```

Figure 5: Prompt template for target harness generation.

incorporates a target description, the entry function, and the corresponding source file. The target description provides an overall summary of the fuzzing objective, which may come from prior vulnerability reports (e.g., CVE records) or expert knowledge indicating potential weaknesses in the target function. This description helps guide the LLM in generating a relevant and effective harness. The entry function is determined based on the structure of the call chain. Given an available call chain $C = (F_n, F_{n-1}, \ldots, F_1, F_0)$, if the final function F_n is the main function, then the entry function is set to F_{n-1}, as it is the actual initiator of the call sequence. The main function is treated as a template for invoking F_{n-1}. In this case, the source file provided as input should be the file containing F_n to ensure that the LLM has all necessary configuration details for calling F_{n-1}. If F_n is not the main function, then F_n itself is treated as the entry function, and the source file corresponds to the file where F_n is defined. By structuring the prompt in this manner, HG-Fuzzer enables LLM to generate a valid fuzz harness that effectively triggers the target function.

Compilation Error Resolution. To ensure that the generated fuzz harness can be successfully compiled into an executable program, HGFuzzer employs a compilation error resolution mechanism

based on RAG (Retrieval-Augmented Generation), as shown in Algorithm 2. This mechanism is based on an external knowledge base that contains all source files, header files, and test cases from the target library. These files are embedded into vector representations and indexed to facilitate efficient similarity-based retrieval. If the harness fails to compile within the library's context, HGFuzzer collects the compilation errors issued by the compiler and constructs a query based on these errors. The query engine retrieves relevant code snippets from the knowledge base, including the usage, implementation, and definition of the error-related functions. The retrieved results are then used by the LLM to guide the repair process. Specifically, the LLM incorporates the context provided by the retrieved snippets to iteratively refine the harness until the compilation errors are resolved. This iterative repair process ensures that the harness is not only syntactically correct but also semantically compatible with the target library.

Algorithm 2 Compilation Error Resolution with RAG.

Require: Compilation error E, Target Harness H, Knowledge base K, Embedding model M, Similarity threshold s, Number of chunks k, LLM L, Refinement prompt P_r

Ensure: Revised harness $H_{revised}$

1: $Q \leftarrow BuildQuery(E)$
2: $IndexBase \leftarrow Embed(K, M)$
3: $Q_{vec} \leftarrow Embed(Q, M)$
4: $C_1, C_2, \ldots, C_k \leftarrow RetrieveChunks(Q_{vec}, IndexBase, s, k)$
5: $R \leftarrow LLM(Q, C_1)$
6: **for all** remaining chunk C_i in $C_2, \ldots, C_k$ **do**
7:      $R \leftarrow RefineWithLLM(R, C_i, P_r, L)$
8: **end for**
9: $H_{revised} \leftarrow LLM(Q, R, H)$
10: **return** $H_{revised}$

3.4 Reachable Input Generation

The target harness generated by HGFuzzer explicitly constrains the fuzzer's path exploration. However, the harness alone cannot directly guide the execution path to the target function. This limitation arises because the entry function often contains multiple branches, which create diverse execution paths. To ensure that the execution path reaches the target function, it is necessary to generate a reachable seed input that satisfies all execution conditions within the call chain. These execution conditions provide logical or mathematical constraints that must be met for the program to traverse the intended path. Given the diversity and complexity of library functionalities and their input formats, generating such an input manually is both labor-intensive and error-prone.

An alternative approach might be to employ LLMs to directly generate the initial input. However, this approach presents significant challenges when considering the complex input requirements of many libraries. For example, libraries such as `libming` process SWF or ABC flash files, `libjpeg-turbo` requires correctly formatted JPEG or PPM images, and other libraries may demand inputs with complex binary structures, specific headers, checksums, and format-specific requirements. Directly generating such specialized

---

Prompt Template

Task: Generate a Python script using standard libraries to create an '*input_file*' whose content precisely satisfies all execution conditions specified below, ensuring the target function is reached via the intended execution path.

Input Context:
- **Target Description**: Information about the fuzzing target ({*target_info*}).
- **Target Function**: The target function to be tested ({*target_func*}).
- **Harness Code**: Code used to run the target with the input ({*harness_code*}).
- **Available Call Chain**: The complete call chain leading to the target function ({*call_chain*}).
- **Execution Conditions**: Execution conditions and dependencies for each function in the call chain ({*execution_conditions*}).

Instructions:
- Satisfy **ALL** execution conditions precisely, prioritizing mathematical/logical correctness over input realism or validity.
- Use conservative values clearly within bounds for range conditions (e.g., `100` if `<200`) and the exact required value for equality conditions (e.g., `127` if `==127`).
- Ensure chosen values strictly follow the specified conditions to avoid triggering alternative execution paths.
- Include detailed comments explaining value choices/condition satisfaction, implement basic error handling, and save the generated input as '*input_file*' with appropriate extension.

Figure 6: Prompt template for reachable input generation.

---

binary or structured data as raw content exceeds the current capabilities of LLMs and would invariably produce invalid inputs. For instance, creating a valid SWF file requires comprehensive knowledge of the flash file format specification, including its tag structure, binary encoding rules, and internal consistency requirements.

To address this challenge, HGFuzzer instead employs LLM to generate a Python script tailored to produce an initial input capable of meeting all specified execution conditions. This approach **leverages existing libraries in Python that can handle the complexities of file format specifications**, allowing for precise construction of valid inputs. To guide the LLM in generating the required Python script, HGFuzzer employs a structured prompt, as listed in Figure 6. The prompt supplies essential input context, encompassing the execution conditions derived from the program's call chain and the pertinent harness code. It explicitly defines the task for the LLM: to generate a Python script producing an input that satisfies all specified execution conditions, thereby ensuring the target function is reached while avoiding unintended execution paths. Additionally, the prompt details how to handle specific condition types, instructing the LLM to use values well within bounds for range constraints and precise values for equality constraints, leveraging Python libraries for accurate input construction.

Once the Python script is generated, it is executed to produce an initial input for the target harness. However, given the potential for the LLM to generate incorrect or suboptimal code [36, 46, 47], HGFuzzer integrates a verification mechanism using afl-cov [1]. This tool instruments the target harness and monitors the execution path to verify whether the generated input successfully guides the program to the target function. If the input fails to reach the target function, the process is repeated iteratively. During each iteration, the prompt is refined, and the LLM generates a new script aiming to address previous deficiencies. This iterative process continues until a reachable input is obtained. By combining the generative capabilities of LLM with the verification of afl-cov, HGFuzzer ensures that the generated inputs effectively guide the execution path to the target function.

3.5 Target-Specific Mutator Generation

After generating a reachable input to guide the execution path to the target function, the next challenge is to ensure that the target function can be triggered. Triggering the target requires the program to enter specific execution states that satisfy the conditions necessary for exploiting the target vulnerability. To reduce the randomness of the fuzz engine in mutating the reachable seed, HGFuzzer leverages LLM to generate **a target-specific custom mutator**. The custom mutator is then integrated into the fuzzing engine to implement tailored mutation strategies designed for the specific target.

The generation of the custom mutator is based on a carefully designed prompt, which incorporates the target description, the target function's source code, and the reachable input script. Additionally, the prompt provides the LLM with the custom mutator API documentation and examples from the fuzzing engine to ensure compatibility. We guide the LLM through a three-step chain-of-thought prompt to generate the custom mutator:

(1) **Analyzing Root Cause**: The LLM first analyzes the target function's source code and the provided description of the target, same as mentioned in Section 3.3. This analysis focuses on identifying the root cause of the vulnerability and the specific program states that must be reached to trigger it. By understanding the vulnerability pattern and its triggering conditions, the LLM gathers critical insights into how the input should be mutated to exploit the vulnerability.

(2) **Designing Mutation Strategy**: Based on the analysis of the target function and its triggering conditions, the LLM designs a mutation strategy aimed at efficiently exploiting the vulnerability. This strategy prioritizes mutations that directly target the vulnerability-triggering conditions. The approach ensures that input values are mutated toward extreme bounds or specific ranges that satisfy the conditions identified in the target function, while simultaneously avoiding mutations that would cause the input to fail the call chain conditions or deviate from paths leading to the target.

(3) **Generating Custom Mutator Code**: After defining the mutation strategy, the LLM generates the complete custom mutator code in C/C++, adhering to the custom mutator API. The mutator code implements the tailored mutation strategy specific to the target vulnerability while ensuring

compatibility with the fuzzing engine. It includes all necessary headers, struct definitions, and initialization functions required by the custom mutator API documentation. The implementation maintains performance optimization to ensure the fuzzing engine runs smoothly without runtime errors or performance degradation.

```
001 METHODDEF(JDIMENSION)
002 get_rgb_row(j_compress_ptr cinfo, cjpeg_source_ptr sinfo)
003 {
004     ppm_source_ptr source = (ppm_source_ptr)sinfo;
005     register JSAMPROW ptr;
006     register U_CHAR *bufferptr;
007     register JSAMPLE *rescale = source->rescale;
008     JDIMENSION col;
009     unsigned int maxval = source->maxval;
010     register int rindex = rgb_red[cinfo->in_color_space];
011     register int gindex = rgb_green[cinfo->in_color_space];
012     register int bindex = rgb_blue[cinfo->in_color_space];
013     register int aindex = alpha_index[cinfo->in_color_space];
014     register int ps = rgb_pixelsize[cinfo->in_color_space];
015
016     if (!ReadOK(source->pub.input_file, source->iobuffer, source->buffer_width))
017         ERREXIT(cinfo, JERR_INPUT_EOF);
018     ptr = source->pub.buffer[0];
019     bufferptr = source->iobuffer;
020     if (maxval == MAXJSAMPLE) {
021         if (aindex >= 0)
022             RGB_READ_LOOP(*bufferptr++, ptr[aindex] = 0xFF;)
023         else
024             RGB_READ_LOOP(*bufferptr++,)
025     } else {
026         if (aindex >= 0)
027             RGB_READ_LOOP(rescale[UCH(*bufferptr++)], ptr[aindex] = 0xFF;)
028         else
029             RGB_READ_LOOP(rescale[UCH(*bufferptr++)],) /* Vulnerable code line */
030     }
031     return 1;
032 }
```

Figure 7: Vulnerable function in CVE-2020-13790.

To ensure the generated custom mutator is functional and effective, our approach incorporates an iterative refinement process. If the generated custom mutator fails to compile or causes runtime errors, the LLM automatically references the custom mutator API documentation to fix the issues. Additionally, our framework performs a lightweight validation check after integrating the custom mutator with the fuzzing engine, measuring basic execution metrics over a short sample run to verify that it operates without significant overhead or instability. This iterative process ensures that the custom mutator can efficiently guide the fuzzing process toward triggering the target vulnerability.

Illustrative Case. To address the heap buffer overflow vulnerability in the get_rgb_row function from CVE-2020-13790 in Figure 7, the LLM first analyzes the source code and identifies that the root cause lies in the lack of proper boundary checks when accessing the rescale array through the pointer bufferptr. This occurs under specific conditions, such as maxval!= MAXJSAMPLE (line 20-25) and aindex<0 (line 26-28), while processing malformed PPM files. Based on this analysis, the LLM designs a mutation strategy that includes generating malformed PPM headers, manipulating key variables (e.g., maxval and aindex), and targeting boundary conditions to trigger the vulnerability. The custom mutator generated by the LLM implements this strategy by mutating input files to include unexpected dimensions, non-standard pixel values, and insufficient data sizes. This allows the fuzzing engine to efficiently

explore paths leading to the vulnerability, reducing randomness and improving the likelihood of triggering the overflow condition.

3.6 Implementation

We implemented HGFuzzer using approximately 3,400 lines of Python code and 500 lines of bash scripts. Our implementation is based on AFL++ [2] due to its extensibility and robust feature set. For our LLM selection, we utilize Anthropic's API to access Claude-3.5-Sonnet [9]. For static analysis, we used CodeQL [3] and Tree-Sitter [7] to extract information about call chains and source code of functions. Additionally, we employed LlamaIndex [5] to build the RAG engine, enabling an efficient construction of prompts and retrieval of relevant program context.

4 Evaluation

In this section, we evaluate HGFuzzer using real-world vulnerabilities and aim to answer the following research questions:

- **RQ1**: How effective is HGFuzzer in triggering known vulnerabilities?
- **RQ2**: Can HGFuzzer effectively constrain execution paths to hit the target functions more frequently?
- **RQ3**: How do the individual components of HGFuzzer contribute to its overall performance?
- **RQ4**: Can HGFuzzer detect new real-world vulnerabilities?

Baselines. For evaluation, we compared HGFuzzer against three state-of-the-art greybox directed fuzzers: AFLGo [10], Beacon [21], and SelectFuzz [39].

- **AFLGo**: A widely used directed fuzzer that guides the fuzzing process toward the target location by leveraging distance metrics to prioritize execution paths.
- **Beacon**: This fuzzer focuses exclusively on execution paths that can reach the target location, prioritizing exploration of relevant code paths.
- **SelectFuzz**: This fuzzer detects code regions related to the target location and focuses on exploring these regions to improve the likelihood of reaching the target.

Since HGFuzzer is built on AFL++, we also included AFL++, a coverage-guided fuzzer, as a baseline to evaluate the effectiveness of our enhancements. We also attempted to compare HGFuzzer with state-of-the-art directed fuzzers such as Titan [22], PGDF [62], and DeepGo [35]. However, Titan is designed for multi-target scenarios and encountered compatibility issues when applied to our single-target benchmark. Additionally, PGDF and DeepGo do not provide source code or documentation, making it infeasible for us to reproduce them.

Benchmark Dataset. We constructed our benchmark dataset by collecting 20 CVE vulnerabilities sourced from prior fuzzing research work [10, 21, 39] and open-source libraries integrated with OSS-Fuzz [6]. These vulnerabilities span 12 versions of 9 open-source C/C++ libraries. As shown in Table 1, the dataset includes common vulnerability types in C/C++ programs and covers diverse functionalities of open-source libraries. These vulnerabilities were used to evaluate the ability of the baselines to perform directed fuzzing on specific targets. To ensure a fair comparison, we used

the same compilation options and execution commands for all experiments. To mitigate the effects of fuzzing randomness on the results, each experiment was repeated 5 times, with a time budget of 24 hours per run.

Environment. All experiments were conducted on a server equipped with an AMD 64-Core Processor and 1TB of RAM, running a 64-bit version of Ubuntu 22.04 LTS.

4.1 RQ1: Effectiveness in Triggering Known Vulnerabilities

To answer RQ1, we evaluated the effectiveness of HGFuzzer against the baselines on the benchmark. We used **the time to exploit (TTE)** as the metric for evaluating effectiveness. Additionally, we compared **the time spent on static analysis (TS)** between HGFuzzer, AFLGo, and Beacon. Since SelectFuzz performs seed execution to detect code regions related to the target during the fuzzing process, and both SelectFuzz and AFL++ only perform compilation and instrumentation during the preparation phase, their TS was not recorded. We also evaluated the time for HGFuzzer to **guide the execution path to the target function (TTR)**. For evaluating the performance of the baselines on the benchmark, we collected the vulnerable library programs mentioned in the CVE reports or the fuzz drivers provided by the libraries as the fuzzing entry points. The initial inputs for the baselines were set to the test inputs or fuzz inputs provided by the libraries.

As detailed in Table 1, **HGFuzzer successfully triggered 17 out of 20 vulnerabilities within the 24-hour time budget**, the highest among all evaluated methods. In contrast, the baselines showed significantly lower success rates, with AFLGo and SelectFuzz each triggering 5 vulnerabilities and Beacon triggering 6, representing a 2.2x to 2.3x improvement in the number of successfully triggered vulnerabilities by HGFuzzer. The baselines also encountered cases where certain CVEs were marked as unavailable due to static analysis errors or runtime failures. For instance, AFLGo and Beacon failed to process CVE-2018-20330 due to block distance calculation errors, while SelectFuzz and AFL++ successfully initiated fuzzing on CVE-2018-20330 but encountered runtime errors, with their execution paths remaining constant at 1. This highlights the robustness of HGFuzzer in handling a wider range of vulnerabilities.

Among the vulnerabilities successfully triggered by HGFuzzer, **11 were exploited within the first minute of fuzzing**, showcasing its efficiency in reaching and triggering vulnerabilities. For example, in CVE-2016-9831, HGFuzzer exploited the vulnerability in under one minute, significantly faster than AFL++'s 0.45 hours, while the TTE for all other baselines exceeded 1 hour for this vulnerability. These results demonstrate that HGFuzzer achieves lower TTEs compared to the baselines, with differences often spanning orders of magnitude. This efficiency is primarily attributed to HGFuzzer's ability to generate precise execution conditions and custom mutators tailored to the target function. By reducing randomness in input mutation and focusing on paths relevant to the vulnerability, HGFuzzer significantly accelerates the fuzzing process.

TS and TTR are influenced by factors such as project size, the complexity of the target function's call relationships, and the response time of the LLM API. However, experimental results show

Table 1: Comparison of effectiveness with baselines on our benchmark. The "Total" row represents the number of vulnerabilities successfully triggered within the time budget.

Library	CVE ID	Vulnerability Location	Vulnerability Type	HGFuzzer			AFLGo		Beacon		SelectFuzz	AFL++
				TS	TTR	TTE	TS	TTE	TS	TTE		
libming 0.4.7	CVE-2016-9831	util/parser.c:parseSWF_RGBA	Buffer Overflow	6.96s	104s	I.E.	2,290s	6.7h	16s	4.3h	1.1h	0.45h
libming 0.4.8	CVE-2017-9988	util/parser.c:parseABC_NS_SET_INFO	NULL Pointer Dereference	6.24s	179s	**10.1h**	3,265s	T.O.	404s	T.O.	T.O.	T.O.
	CVE-2018-13066	util/parser.c:parseSWF_DEFINETEXT	Memory Leak	6.28s	94s	I.E.	3,473s	T.O.	131s	5.8h	T.O.	T.O.
libpng 1.6.34	CVE-2018-13785	pngrutil.c:png_check_chunk_length	Integer Overflow	6.78s	89s	**0.27h**	563s	T.O.	43s	T.O.	T.O.	T.O.
	CVE-2018-14048	png.c:png_free_data	Segmentation Violation	6.79s	96s	T.O.	573s	T.O.	78s	T.O.	T.O.	T.O.
	CVE-2019-7317	pngerror.c:png_safe_execute	Use-after-free	7.12s	107s	I.E.	607s	T.O.	98s	T.O.	T.O.	T.O.
libjpeg-turbo 2.0.1	CVE-2018-20330	turbojpeg.c:tjLoadImage	Buffer Overflow	6.2s	71s	**2.17h**	U.A.	/	U.A.	/	U.A.	U.A.
libjpeg-turbo 2.0.4	CVE-2020-13790	rdppm.c:get_rgb_row	Buffer Overflow	6.25s	77s	**3.12h**	91s	T.O.	22s	T.O.	T.O.	T.O.
libjpeg-turbo 2.0.9	CVE-2021-46822	rdppm.c:get_word_rgb_row	Buffer Overflow	6.52s	82s	T.O.	697s	T.O.	14s	T.O.	T.O.	T.O.
lcms2.9	CVE-2018-16435	src/cmscgats.c:AllocateDataSet	Integer Overflow	6.57s	98s	I.E.	950s	T.O.	82s	T.O.	T.O.	T.O.
libxls 1.4.0	CVE-2017-2896	src/xls.c:xls_mergedCells	Out-of-bounds Write	6.33s	102s	I.E.	41s	I.E.	11s	0.35h	3.7h	0.85h
	CVE-2017-2897	src/ole.c:read_MSAT	Out-of-bounds Write	6.31s	99s	I.E.	55s	0.35h	9s	I.E.	0.25h	0.23h
	CVE-2017-2910	src/xls.c:xls_addCell	Out-of-bounds Write	6.09s	104s	0.13h	64s	1.2h	15s	**I.E.**	0.73h	1.1h
	CVE-2021-27836	src/xls.c:xls_getWorkSheet	Segmentation Violation	6.22s	103s	I.E.	112s	0.45h	16s	0.5h	2.43h	1.23h
libzip 1.2.0	CVE-2017-12858	lib/zip_dirent.c:_zip_dirent_read	Use-after-free	6.12s	82s	**1.1h**	251s	T.O.	14s	T.O.	T.O.	T.O.
libgd 2.3.2	CVE-2021-38115	src/gd_tga.c:read_header_tga	Out-of-bounds Read	7.36s	79s	I.E.	U.A.	/	23s	T.O.	U.A.	T.O.
	CVE-2021-40812	src/gd_bmp.c:_gdImageBmpCtx	Out-of-bounds Read	6.41s	112s	I.E.	543s	T.O.	24s	T.O.	T.O.	T.O.
cjson 1.7.16	CVE-2023-50471	cJSON.c:cJSON_InsertItemInArray	Segmentation Violation	6.59s	69s	I.E.	U.A.	/	U.A.	/	T.O.	T.O.
	CVE-2023-50472	cJSON.c:cJSON_SetValuestring	Segmentation Violation	6.57s	67s	I.E.	11s	T.O.	U.A.	/	T.O.	T.O.
libmodbus 3.1.6	CVE-2024-36844	src/modbus.c:send_msg	Use-after-free	6.09s	98s	T.O.	113s	T.O.	10s	T.O.	T.O.	T.O.
Total				**17/20**			5/20		6/20		5/20	5/20

I.E.: Imiediately Exploit (Trigger the vulnerability in one minute), **T.O.**: Time Out (>24 hours), **U.A.**: Unavailable (Static analysis or runtime error)

Table 2: Comparison of target function hit rates with baselines on our benchmark.

CVE ID	HGFuzzer	AFLGo	Beacon	SelectFuzz	AFL++
CVE-2016-9831	**79.16%(38/48)**	15.71%(383/2,437)	23.21%(13/56)	34.88%(30/86)	14.64%(115/785)
CVE-2017-9988	**53.76%(256/476)**	3.68%(126/3,420)	1.57%(17/1,080)	0%(0/428)	1.39%(73/5,223)
CVE-2018-13066	**81.82%(9/11)**	3.57%(118/3,298)	5.56%(36/647)	0.76%(3/390)	9.48%(452/4,763)
CVE-2018-13785	94.73%(18/19)	94.10%(319/339)	90.09%(100/111)	**97.63%(124/127)**	96.71%(294/304)
CVE-2018-14048	**99.63%(270/271)**	90.15%(119/132)	94.07%(127/135)	96%(120/125)	93.03%(187/201)
CVE-2019-7317	**100%(3/3)**	100.00%(208/208)	100.00%(126/126)	100.00%(58/58)	100.00%(199/199)
CVE-2018-20330	**100.00%(320/320)**	U.A.	U.A.	U.A.	U.A.
CVE-2020-13790	7.94%(29/365)	1.00%(21/2,094)	8.07%(39/483)	**14.10%(22/156)**	4.19%(137/3,264)
CVE-2021-46822	**9.91%(36/363)**	1.03%(16/1,543)	3.07%(18/585)	0%(0/146)	0.74%(20/2,669)
CVE-2018-16435	**22.47%(20/89)**	9.26%(39/421)	2.52%(11/436)	5.77%(3/52)	14.29%(7/49)
CVE-2017-2896	**92.96%(66/73)**	2.42%(8/330)	13.19%(76/576)	4.72%(35/742)	13.69%(66/482)
CVE-2017-2897	**95.83%(69/72)**	83.75%(299/357)	90.84%(119/131)	94.77%(290/306)	79.91%(346/433)
CVE-2017-2910	**56.25%(36/64)**	38.96%(247/634)	30.88%(42/136)	43.99%(267/607)	36.42%(287/788)
CVE-2021-27836	73.53%(50/68)	69.43%(602/867)	73.45%(498/678)	46.87%(1047/2,234)	**75.42%(801/1,062)**
CVE-2017-12858	88.14%(721/818)	90.77%(492/542)	75.70%(349/461)	**99.28%(139/140)**	95.65%(22/23)
CVE-2021-38115	**100.00%(17/17)**	U.A.	**100.00%(5/5)**	U.A.	**100.00%(5/5)**
CVE-2021-40812	**100.00%(13/13)**	44.10%(202/458)	37.70%(204/541)	68.60%(59/86)	42.04%(206/490)
CVE-2023-50471	**80%(8/10)**	U.A.	U.A.	0%(0/114)	0%(0/37)
CVE-2023-50472	**100.00%(3/3)**	0%(0/612)	U.A.	0%(0/102)	0%(0/39)
CVE-2024-36844	**98.75%(79/80)**	4.65%(2/43)	7.14%(3/42)	14.29%(1/7)	18.75%(12/64)
AVG	**64.75%(2,061/3,183)**	18.05%(3,201/17,735)	28.62%(1,783/6,229)	37.22%(2,198/5,906)	15.46%(3,229/20,880)

that **our approach has an advantage in processing time during the preparation phase**. Unlike AFLGo and Beacon, which incur significant computational overhead from calculating block distances during static analysis, **HGFuzzer only requires querying the call chains of the target function within the library**. This results in lower computational overhead and faster processing

speed. For example, in libming, the average static analysis time for AFLGo exceeds 3000 seconds, while for Beacon, it exceeds 183 seconds. In contrast, our approach completes static analysis in just 6–7 seconds. Additionally, HGFuzzer guides the execution path to the target function within an average of **95.6 seconds**, avoiding the need for fuzz region guidance based on path distance calculations.

In conclusion, HGFuzzer effectively balances success rate, efficiency, and computational overhead in triggering known vulnerabilities. It outperforms baselines in both the number of vulnerabilities triggered and the time required to exploit them. Its ability to avoid costly block distance calculations during static analysis further highlights its efficiency, completing preparation phases significantly faster than baselines. These results emphasize HGFuzzer's practicality in triggering known vulnerabilities for open source libraries.

4.2 RQ2: Effectiveness in Constraining Execution Paths

To evaluate whether HGFuzzer can effectively constrain execution paths to hit target functions more frequently, we examined how explicit path constraints generated by the LLM influence the fuzzing process. These path constraints are designed to reduce unnecessary exploration of non-target regions, potentially lowering performance overhead and improving the success rate of triggering vulnerabilities. For our analysis, we need to examine the execution paths explored during the experiments in Section 4.1.

During fuzzing, each unique execution path discovered generates a new seed that is preserved in the fuzzer's queue. Therefore, by analyzing these seeds, we can determine what proportion of the fuzzer's exploration successfully reached our target functions. We compared the target function hit rate across all approaches by using afl-cov to determine whether each generated seed hit the target function, then calculated the overall hit rate. This metric directly measures how effectively each approach guides execution toward the vulnerable code regions.

As illustrated in Table 2, HGFuzzer achieved the highest target function hit rate on 14 out of 20 CVEs. For example, in CVE-2016-9831, HGFuzzer achieved a hit rate of 79.16%, significantly outperforming SelectFuzz (34.88%) and AFL++ (14.64%). Similarly, for CVE-2018-14048, HGFuzzer reached a hit rate of 99.63%, which is higher than SelectFuzz (96%) and other baselines. These results confirm the ability of HGFuzzer to guide execution paths effectively toward target functions. In contrast, the baselines demonstrated very low or even zero hit rates on certain CVEs, such as CVE-2018-13066 and CVE-2023-50472. This suggests that the library programs or their built-in fuzz drivers struggled or failed to reach the target functions, leading to poor performance of the baselines.

HGFuzzer also generated the smallest number of seeds across the benchmark dataset, with a total of 3,183 seeds. This indicates significantly fewer explored paths, demonstrating that **HGFuzzer effectively constrained the execution paths during fuzzing**. Compared to the best-performing baseline, SelectFuzz, which generated 5,906 seeds, HGFuzzer reduced the number of explored paths by 46%. Despite exploring fewer paths, HGFuzzer achieved the highest average target function hit rate of 64.75%, representing a 27.5% improvement over SelectFuzz (37.22%).

These results highlight three key findings: (1) HGFuzzer significantly constrains the execution paths during directed fuzzing, reducing unnecessary exploration of non-target regions and improving efficiency. (2) HGFuzzer achieves the highest target function hit rate, enabling it to fuzz the target functions more frequently, which correlates with its high success rate and low TTE observed in the Section 4.1 experiments. (3) The use of LLM to explicitly

generate target harnesses as fuzzing entry points demonstrates great flexibility, providing a significant advantage even in scenarios where open-source library programs struggle to reach the target functions. This flexibility ensures the effectiveness of HGFuzzer across a diverse range of programs and CVEs.

4.3 RQ3: Ablation Study

HGFuzzer leverages the generative power of LLM to improve multiple key steps in existing directed fuzzing techniques. To evaluate the contributions of each component, we conducted an ablation study by defining three variations of HGFuzzer: (1) HGFuzzer without the reachable input generation component, using the same initial inputs as the baselines in Section 4.1 (**Without Input**); (2) HGFuzzer without the target-specific mutator generation (**Without Mutator**); and (3) HGFuzzer without both reachable input generation and target-specific mutator generation, relying solely on the target harness generation (**Harness-only**). We evaluated the TTE of these variations across the benchmark to measure their effectiveness.

Table 3: Comparison of effectiveness with different HGFuzzer variations.

CVE ID	Without Input	Without Mutator	Harness-only
CVE-2016-9831	0.28h	0.1h	0.25h
CVE-2017-9988	T.O.	T.O.	T.O.
CVE-2018-13066	T.O.	T.O.	T.O.
CVE-2018-13785	9.8h	10.65h	13.7h
CVE-2018-14048	T.O.	T.O.	T.O.
CVE-2019-7317	T.O.	T.O.	T.O.
CVE-2018-20330	5.32h	2.3h	T.O.
CVE-2020-13790	7.65h	9.1h	17.5h
CVE-2021-46822	T.O.	T.O.	T.O.
CVE-2018-16435	T.O.	0.12h	T.O.
CVE-2017-2896	I.E.	I.E.	I.E.
CVE-2017-2897	I.E.	0.16h	0.22h
CVE-2017-2910	I.E.	0.11h	0.15h
CVE-2021-27836	I.E.	0.27h	0.25h
CVE-2017-12858	7.5h	2.45h	7.9h
CVE-2021-38115	I.E.	0.13h	0.05h
CVE-2021-40812	I.E.	0.08h	0.1h
CVE-2023-50471	I.E.	I.E.	I.E.
CVE-2023-50472	I.E.	I.E.	I.E.
CVE-2024-36844	T.O.	T.O.	T.O.
Total	13/20	14/20	12/20

As shown in Table 3, the removal of individual components led to significant degradation in performance. The Without Input variation successfully triggered vulnerabilities in only 13 out of 20 CVEs, with longer TTE compared to the full HGFuzzer (e.g., 5.32h for CVE-2018-20330 compared to 2.17h in Section 4.1). The Without Mutator variation performed slightly better, exposing vulnerabilities in 14 out of 20 CVEs, but its TTE was also noticeably higher (e.g., 10.65h for CVE-2018-13785 compared to 0.27h in Section 4.1). The Harness-only variation showed the weakest performance, successfully exposing vulnerabilities in only 12 out of 20 CVEs, and often failed to reduce TTE to a practical level (e.g., timing out on CVE-2018-20330).

By comparing the results in Table 1 and Table 3, the ablation experiments demonstrate the essential contributions of each component in HGFuzzer. The absence of reachable input generation significantly reduces the ability to guide execution toward the target function, as it ensures the satisfaction of execution conditions derived from the call chain analysis. Similarly, removing the target-specific mutator generation hinders efficient mutation of inputs to satisfy vulnerability-triggering conditions; the fuzzer relies on random mutations, leading to excessive exploration of irrelevant paths and increased TTE (e.g., 5.32h vs. 2.17h in CVE-2018-20330). The Harness-only variation, which lacks both components, performs the worst, as the harness alone cannot ensure correct path traversal or condition satisfaction, leading to lower success rates and frequent timeouts. We observe that for CVE-2023-50471 and CVE-2023-50472, the three variations exhibit no significant differences compared to the results in Section 4.1, as their exploitability heavily depends on specific library execution paths. HGFuzzer effectively captures these features and incorporates them into the target harness, achieving good performance even with the Harness-only variation. These results highlight that **reachable input generation** and **target-specific mutators** are essential to reducing randomness and improving directed fuzzing efficiency, with the full integration of all components providing the best performance.

4.4 RQ4: Vulnerability Detection

In this section, we apply HGFuzzer to detect new vulnerabilities. We selected two open-source libraries from the benchmark dataset (libming and lcms) and tested their latest versions after compilation. Since directed fuzzers require specific fuzzing targets, we adopted two strategies to define these targets. First, we investigated the most recent vulnerabilities in these libraries from the CVE database [4] and used their root cause locations as fuzzing targets, which is based on the observation that certain vulnerable locations may contain multiple related bugs [52]. Second, we invited a security expert to manually review the code of the target libraries to identify additional suspicious locations. We applied HGFuzzer to each target for 24 hours of fuzzing. As a result, HGFuzzer discovered 9 new vulnerabilities across the two open-source libraries. All 9 vulnerabilities were assigned CVE IDs, demonstrating the real-world impact of our approach. The details of these findings are summarized in Table 4.

Table 4: Vulnerabilities identified by HGFuzzer.

Library	Vulnerability Location	Vulnerability Type	CVE
	util/parser.c:parseSWF_EXPORTASSETS	Memory Leak	CVE-2025-26304
	util/parser.c:parseSWF_SOUNDINFO	Memory Leak	CVE-2025-26305
	util/read.c:readSizedString	Memory Leak	CVE-2025-26306
	util/parser.c:parseSWF_IMPORTASSETS2	Memory Leak	CVE-2025-26307
libming 0.4.8	util/parser.c:parseSWF_FILTERLIST	Memory Leak	CVE-2025-26308
	util/parser.c:parseSWF_DEFINESCENEANDFRAMEDATA	Memory Leak	CVE-2025-26309
	util/parser.c:parseABC_FILE	Memory Leak	CVE-2025-26310
	util/parser.c:parseSWF_CLIPACTIONS	Memory Leak	CVE-2025-26311
lcms2.16	cmsgamma.c:smooth2	Buffer Overflow	CVE-2025-29070

To further demonstrate the effectiveness to detect unknown vulnerabilities of HGFuzzer, we provide two representative examples of the vulnerabilities it identified, as shown in Figure 8. First, HGFuzzer identified a memory leak vulnerability in the readSizedStr

```
339 char *readSizedString(FILE *f,int size)
340 {
341     int len = 0, buflen = 256, i;
342     char c, *buf, *p;
343
344     buf = (char *)malloc(sizeof(char)*buflen);
345     p = buf;
346
347     for(i=0;i<size;i++)
348     {
349         c=(char)readUInt8(f);
350         if(len >= buflen-2)
351         {
352             buf = (char *)realloc(buf, sizeof(char)*(buflen+256));
353             buflen += 256;
354             p = buf+len;
355         }
356     }
...     ...
374     return buf;
375 }
```

(a) A memory leak case (CVE-2025-26306) in libming 0.4.8.

```
1137 static cmsBool smooth2(cmsContext ContextID, cmsFloat32Number w[], cmsFloat32Number y[],
1138                         cmsFloat32Number z[], cmsFloat32Number lambda, int m)
1139 {
1140     int i, i1, i2;
1141     cmsFloat32Number *c, *d, *e;
1142     cmsBool st;
1143
1144     c = (cmsFloat32Number*) _cmsCalloc(ContextID, MAX_NODES_IN_CURVE, sizeof(cmsFloat32Number));
1145     d = (cmsFloat32Number*) _cmsCalloc(ContextID, MAX_NODES_IN_CURVE, sizeof(cmsFloat32Number));
1146     e = (cmsFloat32Number*) _cmsCalloc(ContextID, MAX_NODES_IN_CURVE, sizeof(cmsFloat32Number));
1147
1148     if (c != NULL && d != NULL && e != NULL) {
...     ...
1185
1186         i1 = m - 2; i2 = m - 3;
1187
1188         d[m - 1] = w[m - 1] + 5 * lambda -c[i1] * c[i1] * d[i1] - e[i2] * e[i2] * d[i2];
1189         c[m - 1] = (-2 * lambda - d[i1] * c[i1] * e[i1]) / d[m - 1];
1190         z[m - 1] = w[m - 1] * y[m - 1] - c[i1] * z[i1] - e[i2] * z[i2];
1191         i1 = m - 1; i2 = m - 2;
...     ...
1201     }
1202     else st = FALSE;
1203
1204     if (c != NULL) _cmsFree(ContextID, c);
1205     if (d != NULL) _cmsFree(ContextID, d);
1206     if (e != NULL) _cmsFree(ContextID, e);
1207
1208     return st;
1209 }
```

(b) A buffer overflow case (CVE-2025-29070) in lcms2.16.

Figure 8: Examples of identified vulnerabilities.

ing function of libming 0.4.8. The issue arises from insufficient error handling during dynamic memory allocation. The function allocates a buffer using malloc (line 344) and resizes it with realloc (line 352) when the buffer size is exceeded. However, in cases where realloc fails, the previously allocated memory is not freed, leading to a memory leak. Additionally, the function lacks proper cleanup logic in error conditions, such as when readUInt8 fails during the loop (lines 347–356). This vulnerability was assigned CVE-2025-26306. Second, HGFuzzer revealed a buffer overflow vulnerability in the smooth2 function of lcms2. The function dynamically allocates memory for the c, d, and e arrays using _cmsCalloc (lines 1144–1146) and subsequently accesses elements based on the input parameter m. At lines 1188–1190, the function performs calculations that access d[m−1], d[m−2], and d[m−3] without validating whether m is large enough to ensure safe access. If the input m is less than 4, these array accesses result in a buffer overflow.

5 Discussion

HGFuzzer demonstrates the effectiveness of applying LLM to DGF. To explore more effective DGF approaches, we discuss core aspects of our approach and identify promising future directions.

Selection of LLM. In our implementation of HGFuzzer, we utilized Claude-3.5-Sonnet as the underlying LLM. However, the performance of different LLMs may vary significantly, leading to discrepancies in outputs and results. More advanced LLMs generally exhibit a deeper understanding of the instructions provided in prompts and are capable of generating higher-quality harnesses and initial inputs. This suggests that the performance of HGFuzzer could be further enhanced by employing more powerful LLMs. Additionally, we observed during our experiments that hallucinations in the LLM could produce incorrect all-chain analysis results [24]. This issue can lead to the generation of reachable inputs that fail to satisfy specific constraints. For example, in the case of complex conditional branches, as illustrated in Figure 2, the LLM might mistakenly interpret a condition requiring maxval to be less than 255 as requiring maxval to be greater than or equal to 255. Such misinterpretations negatively impact the effectiveness of HGFuzzer, especially in scenarios with intricate logic or constraints. A validation mechanism that cross-checks LLM-generated constraint interpretations against the original code could be implemented to address this issue, which we leave as a direction for future work.

Expanding Applicability. The benchmark libraries used in our evaluation are open-source, with their source code publicly available. These libraries are also likely included in the training data of the selected LLM. As a result, applying HGFuzzer directly to closed-source libraries might yield different evaluation outcomes. Inspired by AFGen's approach of generating harnesses for whole functions of applications [37], we also consider leveraging internal function call relationships as a reference for generating target harnesses. This could enable HGFuzzer to be applied in a broader range of software scenarios, such as Linux-based systems and web frameworks. Expanding the scope of HGFuzzer to these domains would require addressing challenges related to analyzing closed-source code and handling complex system-level dependencies, which presents an interesting direction for future work.

6 Related Work

6.1 Directed Greybox Fuzzing

DGF has recently gained attention due to its ability to efficiently locate specific target sites or trigger certain program behaviors. Böhme et al. [10] proposed AFLGo, a pioneering DGF tool that utilizes a distance-based fitness metric to prioritize seeds closer to target locations. AFLGo calculates target distances during compile time and uses this information at runtime to guide the fuzzing process, achieving significant efficiency improvements in tasks such as bug reproduction. Building on AFLGo, tools like Hawkeye [12] introduced function-level trace similarity to enhance seed prioritization, while tools such as UAFL [49] and UAFuzz [44] focused on detecting complex behavioral bugs like use-after-free vulnerabilities using sequence-aware fitness metrics. Berry [33] extended this concept by incorporating execution context into target sequences, improving the accuracy of target coverage.

To improve target identification, recent tools have integrated advanced techniques. SUZZER [63], V-Fuzz [32], and DeepGo [35] employ deep learning models to predict vulnerable code regions, enabling the automatic labeling of potential targets without manual effort. AFLChurn [65] and DeltaFuzz [61] leverage code changes from version control systems to identify patch-related targets, making them suitable for regression testing scenarios. Additionally, tools like FuzzGuard [67] and BEACON [21] use lightweight static analysis or deep learning to filter out unreachable inputs, significantly improving fuzzing efficiency. For example, FuzzGuard has been shown to reduce unnecessary path executions by over 80%, while BEACON combines symbolic execution with filtering mechanisms to further optimize performance. These advancements highlight the diverse approaches in DGF, focusing on improved target identification, advanced fitness metrics, and optimization strategies to enhance fuzzing performance across a variety of scenarios.

6.2 LLM for Fuzzing

Recent advancements in integrating LLM with fuzzing have demonstrated their potential to address challenges in generating meaningful and context-aware test cases for complex software systems [25, 27, 55]. Unlike traditional fuzzing methods, which often rely on simple input generation techniques, LLM enable more sophisticated approaches by leveraging their generative capabilities. Tools such as TitanFuzz [14], FuzzGPT [15], WhiteFox [58], and ParaFuzz [57] utilize models like GPT [8] and Codex [18] to improve input diversity and quality. These approaches incorporate LLM into prompt engineering and seed mutation processes, enhancing the effectiveness of fuzzing. Furthermore, LLMs have been applied to refine mutation strategies, beyond conventional techniques such as bit-flipping [50]. E.g., CovRL-Fuzz [17] employs LLM to perform coverage-guided mutations while maintaining input validity, increasing the probability of uncovering unexpected software behavior.

In addition to improving fuzzing inputs, LLM are increasingly employed to automate and simplify fuzzing workflows, particularly in the generation of fuzz drivers. InputBlaster [38] demonstrates the use of LLM to create specialized text inputs for mobile applications, achieving higher bug detection rates compared to conventional methods. Similarly, ChatAFL [43] and ChatFuzz [20] integrate LLM to enhance fuzzing for web applications and network protocols, resulting in greater code coverage and better vulnerability identification. LLM have also proven effective in automating fuzz driver generation, which is critical for targeting specific APIs or software functions. Zhang et al. [60] employ GPT-3.5 and GPT-4 to automate the creation of fuzz drivers for complex library APIs, reducing manual effort and achieving over 60% automation. Similarly, CK-GFuzzer [54] leverages LLM along with a code knowledge graph to iteratively generate high-quality fuzz drivers, enabling deeper exploration of previously untested library code and improving code coverage. These advancements demonstrate the potential of integrating LLM with knowledge-driven frameworks to streamline fuzzing processes and enhance their overall effectiveness.

7 Conclusion

In this work, we present HGFuzzer, a novel and automatic framework that integrates LLM to enhance DGF. By transforming path

constraint analysis into code generation tasks, HGFuzzer systematically generates target harnesses and reachable inputs, effectively reducing unnecessary path exploration. Additionally, it employs custom mutators tailored to specific vulnerabilities, minimizing randomness in input mutation and improving fuzzing precision. Evaluated on 20 real-world vulnerabilities, HGFuzzer outperformed state-of-the-art fuzzers, successfully triggering 17 vulnerabilities, 11 of which were triggered within the first minute, and achieving a speedup of at least 24.8× compared to baselines. Moreover, HGFuzzer discovered 9 previously unknown vulnerabilities, all of which received CVE IDs. These results demonstrate the effectiveness of HGFuzzer in improving fuzzing efficiency and precision.

References

[1] 2018. afl-cov. https://github.com/mrash/afl-cov
[2] 2025. AFL++. https://github.com/AFLplusplus/AFLplusplus
[3] 2025. CodeQL documentation. https://codeql.github.com/docs/
[4] 2025. CVE Database. https://cve.mitre.org/
[5] 2025. LlamaIndex Documents. https://docs.llamaindex.ai/en/stable/
[6] 2025. OSS-Fuzz. https://github.com/google/oss-fuzz/tree/master/projects
[7] 2025. Tree-Sitter documentation. https://tree-sitter.github.io/tree-sitter/
[8] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023).
[9] Anthropic. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet
[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (Dallas, Texas, USA) (CCS '17). Association for Computing Machinery, New York, NY, USA, 2329–2344. doi:10.1145/3133956.3134020
[11] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A systematic review of fuzzing techniques. Computers & Security 75 (2018), 118–137. doi:10.1016/j.cose.2018.02.002
[12] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18). Association for Computing Machinery, New York, NY, USA, 2095–2108. doi:10.1145/3243734.3243849
[13] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. 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 (Seattle, WA, USA) (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 423–435. doi:10.1145/3597926.3598067
[14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. 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. 423–435.
[15] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 1–13.
[16] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). 2440–2451. doi:10.1145/3510003.3510197
[17] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 1656–1668. doi:10.1145/3650212.3680389
[18] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In Proceedings of the 24th Australasian Computing Education Conference. 10–19.
[19] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. ACM Trans. Softw. Eng. Methodol. 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988

[20] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. arXiv preprint arXiv:2306.06782 (2023).
[21] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In 2022 IEEE Symposium on Security and Privacy (SP). 36–50. doi:10.1109/SP46214.2022.9833751
[22] Heqing Huang, Peisen Yao, Hung-Chun Chiu, Yiyuan Guo, and Charles Zhang. 2024. Titan : Efficient Multi-target Directed Greybox Fuzzing. 2024 IEEE Symposium on Security and Privacy (SP) (2024), 1849–1864. https://api.semanticscholar.org/CorpusID:268386913
[23] Heqing Huang, Anshunkang Zhou, Mathias Payer, and Charles Zhang. 2024. Everything is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference. In 2024 IEEE Symposium on Security and Privacy (SP). 1956–1973. doi:10.1109/SP54263.2024.00142
[24] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. ACM Trans. Inf. Syst. 43, 2, Article 42 (Jan. 2025), 55 pages. doi:10.1145/3703155
[25] Linghan Huang, Peizhou Zhao, Huaming Chen, and Lei Ma. 2024. Large Language Models Based Fuzzing Techniques: A Survey. arXiv:2402.00350 [cs.SE] https://arxiv.org/abs/2402.00350
[26] Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2023. Benchmarking and Explaining Large Language Model-based Code Generation: A Causality-Centric Approach. arXiv:2310.06680 [cs.SE] https://arxiv.org/abs/2310.06680
[27] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. 2024. When Fuzzing Meets LLMs: Challenges and Opportunities. In Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 492–496. doi:10.1145/3663529.3663784
[28] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (Sacramento, CA, USA) (ASE '24). Association for Computing Machinery, New York, NY, USA, 1408–1420. doi:10.1145/3691620.3695513
[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18). Association for Computing Machinery, New York, NY, USA, 2123–2138. doi:10.1145/3243734.3243804
[30] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu
[31] Tsz-On Li, Wenxi Zong, Yibo Wang, Haoye Tian, Ying Wang, Shing-Chi Cheung, and Jeff Kramer. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). 14–26. doi:10.1109/ASE56229.2023.00089
[32] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. 2022. V-Fuzz: Vulnerability Prediction-Assisted Evolutionary Fuzzing for Binary Programs. IEEE Transactions on Cybernetics 52, 5 (2022), 3745–3756. doi:10.1109/TCYB.2020.3013675
[33] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. 2020. Sequence directed hybrid fuzzing. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 127–137.
[34] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. IEEE Transactions on Reliability 67, 3 (2018), 1199–1218. doi:10.1109/TR.2018.2834476
[35] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, and Kai Lu. 2024. DeepGo: Predictive Directed Greybox Fuzzing. In Proceedings of the Network and Distributed System Security Symposium.
[36] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv:2305.01210 [cs.SE] https://arxiv.org/abs/2305.01210
[37] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. 2024. AFGen: Whole-Function Fuzzing for Applications and Libraries. In 2024 IEEE Symposium on Security and Privacy (SP). 1901–1919. doi:10.1109/SP54263.2024.00011
[38] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Testing the Limits: Unusual Text Inputs Generation for Mobile App Crash Detection with Large Language Model. arXiv:2310.15657 [cs.SE] https://arxiv.org/abs/2310.15657
[39] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In 2023 IEEE Symposium on Security and Privacy (SP). 2693–2707. doi:10.1109/SP46215.2023.10179296

[40] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 3793–3807. doi:10.1145/3658644.3670396

[41] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. doi:10.1109/TSE.2019.2946563

[42] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model guided Protocol Fuzzing. *Proceedings 2024 Network and Distributed System Security Symposium* (2024). https://api.semanticscholar.org/CorpusID:265296188

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

[44] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 47–62. https://www.usenix.org/conference/raid2020/presentation/nguyen

[45] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[46] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. arXiv:2108.09293 [cs.CR] https://arxiv.org/abs/2108.09293

[47] Ruizhong Qiu, Weiliang Will Zeng, Hanghang Tong, James Ezick, and Christopher Lott. 2024. How Efficient is LLM-Generated Code? A Rigorous & High-Standard Benchmark. arXiv:2406.06647 [cs.SE] https://arxiv.org/abs/2406.06647

[48] Wenxuan Shi, Yunhang Zhang, Xinyu Xing, and Jun Xu. 2024. Harnessing Large Language Models for Seed Generation in Greybox Fuzzing. arXiv:2411.18143 [cs.CR] https://arxiv.org/abs/2411.18143

[49] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 999–1010. doi:10.1145/3377811.3380386

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

[51] Pengfei Wang, Xu Zhou, Tai Yue, Peihong Lin, Yingying Liu, and Kai Lu. 2023. The progress, challenges, and perspectives of directed greybox fuzzing. *Software Testing, Verification and Reliability* 34, 2 (Dec. 2023). doi:10.1002/stvr.1869

[52] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. *Proceedings 2020 Network and Distributed System Security Symposium* (2020). https://api.semanticscholar.org/CorpusID:211268394

[53] Yi Xiang, Xuhong Zhang, Peiyu Liu, Shouling Ji, Hong Liang, Jiacheng Xu, and Wenhai Wang. 2024. Critical Code Guided Directed Greybox Fuzzing for Commits. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 2459–2474. https://www.usenix.org/conference/usenixsecurity24/presentation/xiang-yi

[54] Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. 2024. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. arXiv:2411.11532 [cs.SE] https://arxiv.org/abs/2411.11532

[55] Hanxiang Xu, Shenao Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and Haoyu Wang. 2024. Large Language Models for Cyber Security: A Systematic Literature Review. arXiv:2405.04760 [cs.CR] https://arxiv.org/abs/2405.04760

[56] Yijiang Xu, Hongrui Jia, Liguo Chen, Xin Wang, Zhengran Zeng, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, Shikun Zhang, and Zhonghai Wu. 2024. ISC4DGF: Enhancing Directed Grey-box Fuzzing with LLM-Driven Initial Seed Corpus Generation. arXiv:2409.14329 [cs.SE] https://arxiv.org/abs/2409.14329

[57] Lu Yan, Zhuo Zhang, Guanhong Tao, Kaiyuan Zhang, Xuan Chen, Guangyu Shen, and Xiangyu Zhang. 2024. Parafuzz: An interpretability-driven technique for detecting poisoned samples in nlp. *Advances in Neural Information Processing Systems* 36 (2024).

[58] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *arXiv preprint arXiv:2310.15991* (2023).

[59] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2139–2154. doi:10.1145/3133956.3134085

[60] Cen Zhang, Mingqiang Bai, Yaowen Zheng, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2023. Understanding large language model based fuzz driver generation. *arXiv e-prints* (2023), arXiv–2307.

[61] Jia-Ming Zhang, Zhan-Qi Cui, Xiang Chen, Huan-Huan Wu, Li-Wei Zheng, and Jian-Bin Liu. 2022. DeltaFuzz: historical version information guided fuzz testing. *Journal of Computer Science and Technology* 37, 1 (2022), 29–49.

[62] Yujian Zhang, Yaokun Liu, Jinyu Xu, and Yanhao Wang. 2024. Predecessor-aware Directed Greybox Fuzzing . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1884–1900. doi:10.1109/SP54263.2024.00040

[63] Yuyue Zhao, Yangyang Li, Tengfei Yang, and Haiyong Xie. 2020. Suzzer: A Vulnerability-Guided Fuzzer Based on Deep Learning. In *International Conference on Information Security and Cryptology*.

[64] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1633–1644.

[65] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2169–2182.

[66] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (Sept. 2022), 36 pages. doi:10.1145/3512345

[67] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 127, 15 pages.