PromeFuzz: A Knowledge-Driven Approach to Fuzzing Harness Generation with Large Language Models

Yuwei Liu*
Institute of Software Chinese Academy of Sciences
Ant Group
Beijing, China
lyw458372@antgroup.com

Junquan Deng*
Institute of Software Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
dengjunquan2024@iscas.ac.cn

Xiangkun Jia†
Institute of Software Chinese Academy of Sciences
Beijing, China
xiangkun@iscas.ac.cn

Yanhao Wang
Independent
Beijing, China
wangyanhao136@gmail.com

Minghua Wang
Ant Group
Beijing, China
minghua.wmh@antgroup.com

Lin Huang
Ant Group
Beijing, China
linyu.hl@antgroup.com

Tao Wei
Ant Group
Beijing, China
lenx.wei@antgroup.com

Purui Su
Institute of Software Chinese Academy of Sciences
Beijing, China
purui@iscas.ac.cn

Abstract

API-level fuzzing has become increasingly important for discovering subtle bugs in modern software, yet generating effective fuzzing harnesses remains a complex and error-prone task. Existing approaches often rely on limited consumer code or shallow program analysis, which fail to capture deep API semantics and interdependencies, resulting in poor coverage and high false positive rates. Recent methods incorporating Large Language Models (LLMs) have improved harness generation by leveraging pretrained knowledge, but they still struggle with hallucinations and lack domain-specific understanding.

To address the challenges, we present PromeFuzz, a knowledge-driven framework for automatic fuzzing harness generation using LLMs. PromeFuzz constructs a structured knowledge base by combining code metadata, API documentation, and real-world call correlations to enhance the semantic accuracy and coverage of generated harnesses. It further integrates retrieval-augmented generation and a dedicated sanitizer module to refine harness quality and triage crashes. We evaluate PromeFuzz on 22 open-source projects, demonstrating significant improvements over state-of-the-art tools. Specifically, PromeFuzz achieves 1.50×, 3.88×, 1.91× and 1.40× higher branch coverage than 3 LLM-based baselines (PromptFuzz, CKGFuzzer and OSS-Fuzz-Gen) and manually crafted harnesses (OSS-Fuzz), respectively. It also discovers more unique crashes with 89.7% precision and uncovers 25 previously unknown vulnerabilities (21 confirmed by the developer and 3 assigned with CVE IDs).

Keywords

Fuzzing, API-level Fuzzing, Fuzzing Harness Generation, Large Language Model

1 Introduction

Fuzzing has long been recognized as an effective technique for uncovering security vulnerabilities by automatically generating and executing a diverse set of inputs [2, 3, 6, 8, 14, 15, 18, 20, 24, 26, 30, 32, 34, 46, 52, 53, 55, 58]. Traditional fuzzing tools primarily mutate inputs and monitor crashes, memory leaks, or other abnormal behaviors in program execution. However, with the increasing significance of APIs in modern applications and libraries, API-level fuzzing has emerged as a critical testing paradigm [42]. Tools like libFuzzer [41] allow testers to focus on individual API functions and their interdependencies, increasing the likelihood of uncovering subtle bugs that might be missed by traditional program fuzzing.

Unlike traditional fuzzing—which involves not only generating seed inputs but also employing various techniques to explore full program behavior—API-level testing introduces additional complexities. In this context, it is not enough to simply generate diverse inputs; the test harness itself must be logically sound, adhere to programming conventions, and successfully compile and execute. Essentially, constructing a fuzzing harness for an API presents challenges similar to building a consumer (e.g., a program that utilizes the target library or its unit tests). The harness must satisfy three key requirements: (1) it must conform to the syntactic and initialization constraints necessary for a compilable program, (2) it must adhere to the semantic rules of the API (including proper function combinations, call order, and implicit constraints), and (3) it must effectively explore the program's state space.

Existing approaches predominantly rely on existing consumer code to learn API usage patterns. For instance, FUDGE [2] slices

*Both authors contributed equally to this research.
†Corresponding author.

consumer code to maintain syntactic correctness, while AFGen [31], FuzzGen [18], APICraft [55], UTopia [19], and RULF [20] extract call sequences and usage semantics from consumer programs or unit tests to generate harnesses. Although these methods leverage valuable consumer-provided information to ensure both syntactic and semantic correctness, they are constrained by limited knowledge. As a result, they often fail to represent complex inter-function relationships and implicit constraints between API calls—especially when there are few or no useful consumer examples available. Moreover, while static analysis is sometimes employed to expand program state exploration by modifying API sequences [5, 15, 27, 54], it may unfortunately bring negative benefits, e.g., disrupting subtle interdependencies that static analysis fails to detect, and leading to spurious errors and high false-positive rates. This, in turn, results in incomplete test suites and limited code coverage.

Recent breakthroughs have integrated Large Language Models (LLMs) into harness generation for API testing, leveraging their embedded knowledge to better explore program states while striving to ensure both syntactic and semantic correctness. Approaches such as PromptFuzz [35] and OSS-Fuzz-Gen [28] analyze API signatures—including function names and parameter types—to generate diverse test cases with correct semantics, significantly reducing manual effort. Similarly, methods like CKGFuzzer [48], which utilize abstract syntax tree (AST) information, enable LLMs to infer relationships among APIs and produce more coherent test sequences. However, these methods often suffer from limited domain-specific knowledge, which hinders their ability to capture subtle API usage patterns. For example, without adequate knowledge from documentation, LLMs struggle to infer implicit constraints and interrelationships among API functions, which causes numerous hallucinations and failures when handling complex tasks. Consequently, they sometimes produce false positives and tend to overlook crucial harness-level stages like automated compilation fixes, harness validation, and crash analysis.

Motivated by these challenges, our method, PromeFuzz, builds on the strengths of LLM while trying to address its shortcomings. At its core, our approach transforms all project-related information—including source code and documentation—into structured knowledge that LLMs can leverage more effectively. By integrating this knowledge-driven foundation with the LLM's capability to perform well-defined tasks, we have developed an automated solution for harness generation and fuzz testing.

Specifically, we construct a rich knowledge base by combining three critical categories of information: (1) Code Metadata Knowledge: We employ program analysis to extract API function metadata and associated data types, constructing a dependency graph that guarantees the syntactic and initialization requirements needed for compilable harness generation. (2) Documentation Knowledge: By analyzing API implementations and documentation, we capture both explicit constraints (e.g., API contracts and preconditions) and implicit constraints (e.g., usage patterns and comments), thereby enhancing semantic correctness. (3) API Correlation Knowledge: We utilize real-world call sequences in existing consumer code to extract API correlation metrics and further guide the meaningful combination of API functions. Augmented by real-time coverage feedback, this knowledge enables the generation of test sequences that more thoroughly explore diverse program states. Furthermore, we enhance our knowledge base by employing Retrieval-Augmented Generation (RAG), which helps to refine the knowledge available to the LLM. A dedicated sanitizer module based on LLMs is also incorporated to analyze crash reports, determine root causes, and filter out false positives. Together, these components form an end-to-end automated solution that not only integrates traditional static and dynamic analysis methods but also incorporates the advanced capabilities of LLMs for improved API harness generation.

We implement a prototype tool of PromeFuzz based on the concept of knowledge-driven fuzzing harness generation. To evaluate its effectiveness, we conducted experiments on 22 open-source projects to generate fuzzing harnesses for their API functions. The experimental results demonstrate that PromeFuzz outperforms other LLM-based fuzzing harness generation methods (i.e., PromptFuzz, CKGFuzzer and OSS-Fuzz-Gen) as well as manually crafted harnesses from OSS-Fuzz in two key metrics: branch coverage and bug discovery. Specifically, PromeFuzz achieved 1.50× more branch coverage than PromptFuzz, 3.88× more than CKGFuzzer, 1.91× more than OSS-Fuzz-Gen, and 1.40× more than OSS-Fuzz. Additionally, PromeFuzz found more unique crashes than PromptFuzz, CKGFuzzer, OSS-Fuzz-Gen and OSS-Fuzz, and achieved a precision of 89.7%, surpassing PromptFuzz, CKGFuzzer and OSS-Fuzz-Gen. During the experiments, PromeFuzz discovered 25 previously unknown vulnerabilities (21 confirmed and 3 assigned with CVE ID).

In summary, this paper makes the following contributions.

- We propose a novel knowledge-driven approach for fuzzing harness generation with knowledge of code metadata, documentation, and API correlation, which enables LLM-generated harnesses to explore more program state space while keeping syntactically and semantically correct.
- We implement and open-source PromeFuzz[1], a framework that combines knowledge-driven harness generation with automated compilation fixes, coverage feedback, and a dedicated sanitizer to improve the fuzzing coverage and filter false positives.
- We evaluate PromeFuzz on 22 real-world projects. The experiment results show significant improvements in branch coverage (1.50× over PromptFuzz, 3.88× over CKGFuzzer, 1.91× over OSS-Fuzz-Gen, and 1.40× over OSS-Fuzz) and bug discovery (with 25 new vulnerabilities and 89.7% precision, including 21 confirmed bugs and 3 assigned CVEs).

2 Background and Motivation

2.1 Fuzzing with Harness

Fuzzing is a widely adopted technique for discovering vulnerabilities and is commonly used in testing both applications and libraries. While traditional application fuzzing typically starts from the `main` function, fuzzing libraries requires the creation of specialized harnesses that provide the necessary context to invoke API functions correctly. A fuzzing harness must (1) satisfy syntactic constraints (e.g., correct data types and initialization sequences), (2) adhere to semantic rules (e.g., valid call orders and implicit dependencies between functions), and (3) aim to maximize the state space exploration of the target library.

[1] https://github.com/pvz122/PromeFuzz

```
1  // Signature:
2  pcap_t *pcap_create(const char *device, char *errbuf);
3  int pcap_activate(pcap_t *p);
4  int pcap_fileno(pcap_t *p);
5  int pcap_set_protocol_linux(pcap_t *p, int protocol);
6  int bpf_validate(const struct bpf_insn * f, int len);
7  int pcap_minor_version(pcap_t *p);
8
9  int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
10     if(size<0) return 0;
11     // Step 0: Initialize resources
12     FILE *input_file_ptr = fopen("input_file", "wb");
13     if (input_file_ptr == NULL) {return 0;}
14     fwrite(data, sizeof(uint8_t), size, input_file_ptr);
15     fclose(input_file_ptr);
16     const char* input_file = "input_file";
17     // Step 1: pcap_fileno
18     pcap_t* pcap = pcap_create(input_file, NULL);
19     pcap_activate(pcap);
20     int pcap_fd = pcap_fileno(pcap);
21     // Step 2: pcap_set_protocol_linux
22     pcap_set_protocol_linux(pcap, pcap_fd);
23     // Step 3: bpf_validate
24     struct bpf_insn bpf_program[1];
25     int bpf_len = sizeof(bpf_program) / sizeof(bpf_program[0]);
26     int bpf_valid = bpf_validate(bpf_program, bpf_len);
27     // Step 4: pcap_minor_version
28     int minor_version = pcap_minor_version(pcap);
29     // Step 5: Release resources
30     pcap_close(pcap);
31     return 0;
32 }
```

Listing 1: A fuzzing harness for `libpcap` from PromptFuzz.

In practice, researchers often manually construct fuzzing harnesses to continuously test libraries, such as in the Google OSS-Fuzz Project [42]. However, manually written harnesses usually cover only a small subset of available API functions, and many libraries lack any harness at all. Existing automated approaches—which rely on static analysis of consumer code or unit tests [2, 18–20, 31, 49, 55]—often struggle when usage examples are scarce. These methods may also violate semantic constraints by modifying API sequences without fully understanding implicit dependencies.

Recent research has explored using large language models (LLMs) to address these challenges, leveraging their embedded knowledge to improve coverage while maintaining syntactic and semantic correctness [28, 35, 48]. However, existing LLM-based methods often fail to fully utilize information from both the code base and the accompanying documentation. As a result, they tend to generate low-quality harnesses, which not only produce high false positive rates but also limit the effective exploration of a library's logical state space. We illustrate the limitations of these existing approaches through a motivating example.

2.2 Motivation Example

Listing 1 presents a fuzzing harness for `libpcap` that was generated by PromptFuzz [35], a state-of-the-art LLM-based method. Using a random selection strategy, PromptFuzz targets 6 API functions within `libpcap` and takes only the function signatures and type definitions (lines 2–7) as input to the LLM. The harness begins by initializing the necessary resources for fuzzing—writing random data to a temporary file (lines 12–16) and subsequently loading this file via `pcap_create` (line 18). It then processes the loaded data through various `libpcap` API functions (lines 20–28), aiming to maximize code coverage while ensuring syntactic and semantic correctness. Finally, all allocated resources are carefully released (lines 29–30).

However, the harness crashes at line 18 because it calls `pcap_create` with a `NULL` memory buffer. Although the authors of PromptFuzz reported this issue on the GitHub repository[2] and marked it as *Confirmed* in their paper, the crash is actually a false positive caused by API misuse. According to the `libpcap` documentation[3], the `errbuf` parameter must be a buffer large enough to hold at least `PCAP_ERRBUF_SIZE` characters, and thus cannot be `NULL`. This requirement is further underscored by the developer during issue discussions. Similarly, the harness produced by CKGFuzzer [48], a more recent LLM-based approach, exhibits the same false positive. In fact, both PromptFuzz and CKGFuzzer suffer from significant false positive issues, with their precision rates reported at only 1.8% and 6.3%, respectively, as detailed in Section 5.2.

2.3 Challenges

Our analysis indicates that the false positive in the above example arises because PromptFuzz only acquires partial code knowledge, neglecting vital documentation and additional insights from the target library. Although LLMs are capable of extracting some semantic relationships directly from the source code (e.g., the constraint between `bpf_program` and `bpf_len` in line 25), they do not capture the full spectrum of information necessary for reliable harness generation. This shortfall leads to issues such as false positives and inadequate exploration of the library's logical behavior. To overcome these shortcomings, we propose leveraging the complete knowledge of the target library, which introduces the following challenges.

Challenge 1: *How can we effectively extract knowledge from the target library?* Generating high-quality fuzzing harnesses depends on comprehensive knowledge drawn from both the library's source code and its documentation. While PromptFuzz extracts function signatures and type definitions, and CKGFuzzer further employs CodeQL [1] to construct call graphs, relying solely on code knowledge proves inadequate. Retrieval-Augmented Generation (RAG) has been successfully applied to documentation [22, 33, 38, 56], yet its dependence on textual similarity metrics limits its ability to capture the structural and semantic nuances of the code. To bridge this gap, it is critical to identify which pieces of information—such as the relationships among `libpcap` APIs, data structures, function signatures, and constraint-related documentation—are essential for harness generation. In this approach, program analysis techniques are used to extract precise, structured knowledge from the code, while LLM-based methods complement this by interpreting semantic content from both the code and documentation.

Challenge 2: *How can we utilize the extracted knowledge to improve fuzzing harness quality?* The additional knowledge can enhance harness quality, reduce false positives and expand the exploration of program states. However, integrating too much information into the LLM input can trigger the "lost-in-the-middle" phenomenon [29], degrading the output quality. If the input exceeds the model's context window (e.g., GPT-4's 128K or DeepSeek-V3's

[2] https://github.com/the-tcpdump-group/libpcap/issues/1239
[3] https://www.tcpdump.org/manpages/pcap_create.3pcap.html

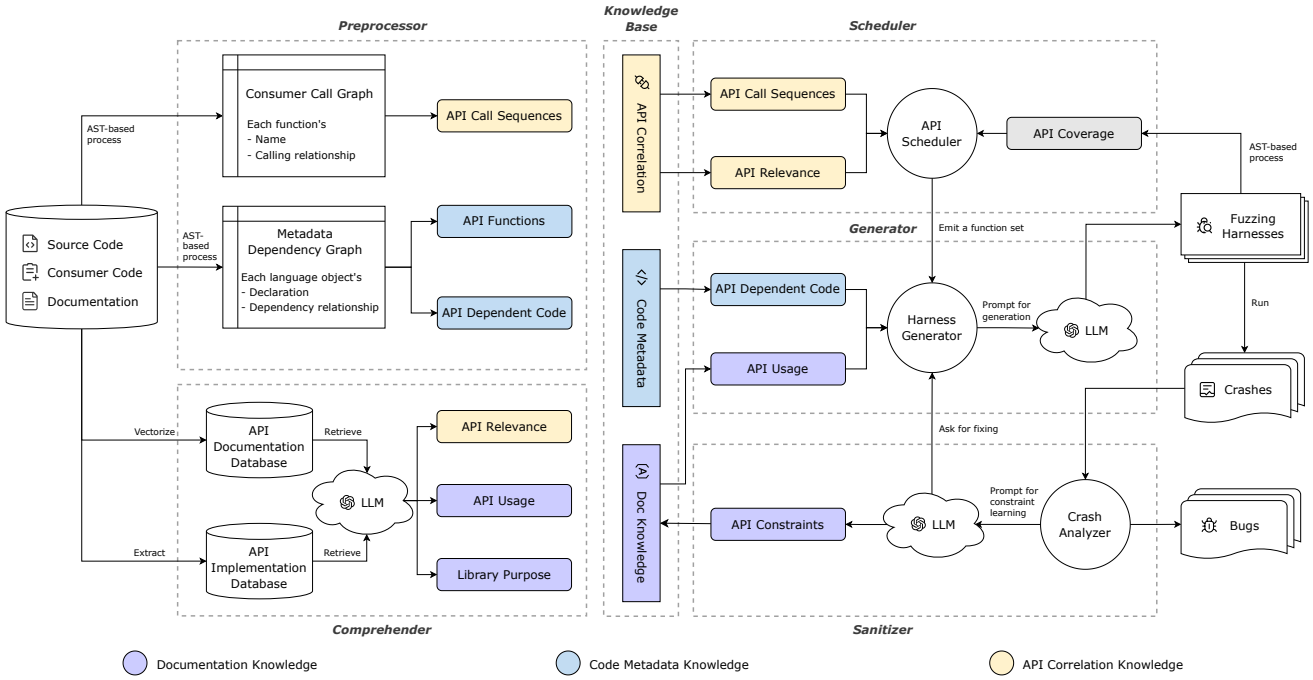


Figure 1: The workflow of PromeFuzz.

64K token limit), it can completely inhibit valid harness generation. For instance, CKGFuzzer's attempt to incorporate an entire call graph for the target library led to input size issues in 12 out of 22 tested libraries, preventing harness generation as shown in Table 2. Thus, it becomes necessary to partition the harness generation process and selectively provide relevant knowledge at each stage.

Challenge 3: *How can we systematically handle mistakes made by LLMs?* Despite their impressive code generation capabilities, LLMs still frequently make mistakes, such as failing to follow instructions or generating hallucinations. This remains a significant challenge for current approaches [28, 35, 48]. For example, PromptFuzz simply discards harnesses that lead to compilation or runtime errors, relying instead on the LLM's stochastic nature to eventually produce valid ones. This strategy not only increases query cost but also contributes to a high false positive rate. Therefore, it is imperative to develop systematic techniques to detect, correct, and learn from such mistakes, ultimately improving robustness and efficiency.

2.4 Overview

To address the above challenges, we propose PromeFuzz, a comprehensive framework for collecting, processing, and utilizing knowledge to generate fuzzing harnesses. It features a feedback mechanism that enhances the knowledge base during the generation process. As shown in Figure 1, PromeFuzz is divided into five modules: *Preprocessor*, *Comprehender*, *Scheduler*, *Generator*, and *Sanitizer*. Knowledge is marked with different colors to highlight its usage across the different modules. For **Challenge 1**, we identify three critical types of knowledge that should be considered for fuzzing harness generation: code metadata, documentation knowledge, and API correlation[4]. Code metadata encompasses the relevant source code involved in API invocation, such as structures requiring initialization and class constructors that need to be called. Documentation knowledge refers to the API usage descriptions provided in the library documentation, which often detail various usage scenarios and implicit rules. API correlation pertains to the relationships between API functions and serves as the basis for selecting and ordering API function invocations. For **Challenge 2**, PromeFuzz transforms code metadata, documentation, and API correlation into structured knowledge and constructs a knowledge base. During the generation process, only metadata and constraints dependent on the target API are selected. For **Challenge 3**, we employ a reasoning model-based crash analyzer in *Sanitizer* to validate harnesses and analyze crashes.

By integrating aforementioned methodology, PromeFuzz effectively eliminates the false positives demonstrated in Section 2.2 during the generation phase. As shown in the comprehension prompt for `pcap_create` in `libpcap` at Figure 3 in Appendix B, the constraint of `errbuf` is extracted from the library documentation. Eventually, with the assistance of later stages, a reliable and effective harness for `libpcap` is constructed.

2.5 Our Approach and Novelty

Prior works have made significant processes in automated harness generation, yet they often struggle with semantic correctness and state space exploration due to an incomplete understanding of the target library. Our approach, PromeFuzz, distinguishes itself by

[4]We use three background colors to represent three types of knowledge.

Table 1: A brief comparison with related works. (✓✗denotes incomplete or unpractical.)

Related Works	Program Analysis	Semantic Comprehension	API Scheduling	Harness Sanitizing	Constraint Learning	Crash Triage
PromptFuzz	✗	✗	✓✗(Usage code + coverage)	✗	✗	✗
CKGFuzzer	✓✗(Call graph)	✗	✓✗(Usage code + coverage)	✓✗(Build error)	✗	✓✗(Not work)
OSS-Fuzz-Gen	✗	✗	✓✗(Coverage)	✓✗(Build error)	✗	✗
PromeFuzz	✓	✓	✓	✓	✓	✓

systematically emulating the process of gathering knowledge and reasoning of a human expert. We achieve this through four key methodological advancements that directly address the challenges identified earlier.

First, for deeper syntactic and structural understanding (Challenge 1), PromeFuzz moves beyond simple API signatures of the previous works and builds a fine-grained metadata dependency graph that precisely models how API-related data structures must be initialized and manipulated, providing a stronger foundation for effective harness generation.

Second, we enrich this structural knowledge with semantic context by systematically distilling insights from documentation using RAG, whereas prior approaches largely overlook this valuable resource. By capturing documented usage constraints, we can prevent the kinds of API misuse illustrated in our motivating example.

Third, to generate more logical and effective harnesses (Challenge 2), we introduce a more sophisticated API scheduling strategy. Instead of relying solely on consumer code or basic coverage feedback, PromeFuzz jointly considers API relevance, consumer usage patterns, and runtime feedback, enabling the creation of more meaningful and complex API interaction sequences.

Finally, and most importantly, PromeFuzz introduces a closed-loop, self-correcting mechanism to handle errors (Challenge 3). Instead of discarding failing harnesses or performing only basic build checks, it employs a novel crash analyzer that sanitizes harnesses to reduce false positives and learns from failures, enabling the long-term knowledge accumulation. This is a unique feature absent in previous approaches.

Together, these integrated techniques enable a more accurate, robust, and knowledge-aware fuzzing harness generation process. A detailed quantitative comparison with existing tools is presented in Table 1 and the evaluation section.

3 PromeFuzz Design

In this section, we present the design details of the five main modules of PromeFuzz: *Preprocessor*, *Comprehender*, *Scheduler*, *Generator*, and *Sanitizer*. The overall workflow of PromeFuzz is shown in Figure 1, and concrete examples illustrating each step are provided in Figure 7 in Appendix A.

3.1 Knowledge Base Construction

PromeFuzz constructs its knowledge base by extracting and comprehending the three types of knowledge from the target library, as illustrated in the left side of Figure 1.

3.1.1 Preprocessor. The *Preprocessor* module processes the entire code base, encompassing both the library source code and consumer source code, to extract essential information for subsequent analysis. It employs specialized algorithms on Abstract Syntax Tree (AST) to extract code metadata, detailed in Section 4.

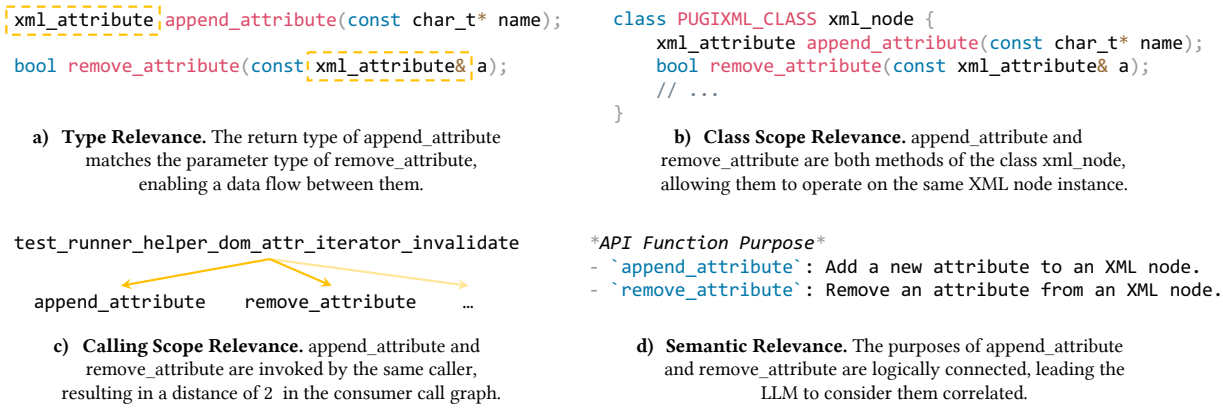
Metadata Dependency Graph. The *Preprocessor* module first constructs a metadata dependency graph, which is an undirected graph of language objects with edges representing various types of dependency relationships. It parses all language objects within the library source files and headers, including functions, structures, classes, enums, and typedefs. It extracts their attributes, declarations, and, most importantly, their dependencies on other language objects. A dependency represents the relationship where one language object references or is a member of another. As illustrated in Figure 7 in Appendix A, the API function signature `xml_node::remove_attribute(xml_attribute& a)` from the `pugixml` library indicates dependencies on both the `xml_node` class and the `xml_attribute` structure. These relationships collectively form an undirected dependency graph, which is used to identify API-dependent code-the language objects that API functions rely on. By obtaining their declaration code, we acquire the necessary code knowledge for invoking an API function.

Consumer Call Graph. For library consumers—such as usage examples, unit tests, and existing fuzzing harnesses—the *Preprocessor* extracts call statements and constructs a call graph. The API call sequences within this call graph provide practical insights into API invocation patterns. These sequences serve as valuable supplements to textual usage descriptions by offering concrete examples, which are subsequently processed in the *Comprehender*.

3.1.2 Comprehender. The *Comprehender* is designed to synthesize knowledge by integrating two key components: a RAG module that extracts API usage patterns from library source code and documentation, and an API correlation measurement module that evaluates the relevance among API functions.

Database Setup. For API source code, implementations are extracted and categorized by API names using AST analysis, forming an API implementation database. For documentation, the content is segmented into semantically coherent chunks, which are then encoded via an embedding model to construct a vector database.

API Usage Comprehension. To comprehend a specific API function, we retrieve the databases using the API name to get relevant code snippets and documentation chunks. The LLM then processes the results and generates the summary of the API usage. For instance, Figure 7 in Appendix A illustrates a concrete case of API comprehension. Figure 3 in Appendix B shows the detailed prompt used for the motivation example involving the `pcap_create` function from the `libpcap` library. As evident from the LLM's response, it successfully identified the key constraint



a) Type Relevance. The return type of append_attribute matches the parameter type of remove_attribute, enabling a data flow between them.

b) Class Scope Relevance. append_attribute and remove_attribute are both methods of the class xml_node, allowing them to operate on the same XML node instance.

c) Calling Scope Relevance. append_attribute and remove_attribute are invoked by the same caller, resulting in a distance of 2 in the consumer call graph.

d) Semantic Relevance. The purposes of append_attribute and remove_attribute are logically connected, leading the LLM to consider them correlated.

Figure 2: Examples for different types of relevance.

that PromptFuzz failed to capture—specifically, the required length of the `errbuf` buffer.

Furthermore, our RAG-based approach extends beyond individual API functions to comprehend the library purpose, providing essential context for understanding API invocations during subsequent generation phases.

API Correlation Measurement. API functions within a library are often interdependent, with certain functions designed to be used together. Figure 2 illustrates two related APIs in the XML document processing library `pugixml`. The `append_attribute` function adds an attribute to an XML node, while `remove_attribute` removes an attribute. This functional pairing demonstrates how combined API usage can both reveal interface constraints and enable deeper state exploration during fuzzing tests. We refer to this correlation knowledge as API relevance and assess it from three dimensions: type relevance, scope relevance, and semantic relevance. Previous approaches typically infer API relationships based on parameter and return types [15, 18] or from usage patterns in consumer code [2, 31]. Inspired by these methods, we incorporate the semantic understanding capabilities of LLMs to develop a three-level API relevance metric covering type, scope, and semantic relevance.

- *Type Relevance.* Type relevance measures the compatibility between parameter types and return types among API functions, which indicates the potential data flow. As shown in Figure 2a, `append_attribute` returns an `xml_attribute` object while `remove_attribute` accepts a reference to `xml_attribute` as the parameter. This compatibility allows the return value of `append_attribute` to be directly passed as the parameter to `remove_attribute`, indicating that these two APIs can be used together. To quantify this relationship mathematically, for two API functions f_p and f_q, let $type\_cnt(f)$ denote the count of return and parameter types; $comm\_cnt(f_p, f_q)$ denote the number of common types shared between two APIs. We define their type relevance as:

$$typ\_rel_{pq} = RMS\left(\frac{comm\_cnt(f_p, f_q)}{type\_cnt(f_p)}, \frac{comm\_cnt(f_p, f_q)}{type\_cnt(f_q)}\right) \quad (1)$$

where $RMS = \sqrt{\frac{a^2+b^2}{2}}$ represents the root mean square.

- *Scope Relevance.* Scope relevance measures the degree to which API functions interact within shared execution contexts. For instance, in object-oriented programming, a class defines its own class scope for its members, where the API functions share the internal states of the object and exhibit strong interdependence. As illustrated in Figure 2b, `append_attribute` and `remove_attribute` are both members of the class `xml_node`, enabling them to operate on the same XML node instance. We refer to this type of relevance as class scope relevance, and define it as follows:

$$class\_scp\_rel_{pq} = \begin{cases} 1 & \text{if belong to the same class} \\ 0 & \text{if not belong to the same class} \end{cases} \quad (2)$$

Another type of scope relevance is calling scope relevance. In consumer code, a calling scope contains sequential API invocations and represents real-world data and control flow interactions among the APIs, demonstrating their mutual relevance. The intuition to define the calling scope is that in the consumer call graph, the closer two API functions are, the more likely they are to be related to each other. Here, we model the consumer call graph as an undirected graph, where the distance of the shortest path from function f_p to function f_q is denoted as $dis(f_p, f_q)$. The calling scope relevance between these two functions then can be calculated as:

$$calling\_scp\_rel_{pq} = \frac{1}{dis(f_p, f_q) - 1} \quad (3)$$

For example, Figure 2c demonstrates that when the two API functions `append_attribute` and `remove_attribute` are both called by the same `pugixml` test function (yielding a distance of 2), they are considered most relevant, achieving a maximum calling scope relevance of 1. Conversely, if two API functions are never called by the same caller node, the distance will be infinite, and their relevance will be 0. If there are multiple consumer code, we will keep the largest calling scope relevance value.

- *Semantic Relevance.* Semantic relevance measures the functional relationships between APIs based on their design purposes. Figure 2d shows the purposes of the APIs `xml_node::append_attribute` (to add a node attribute) and `xml_node::remove_attribute` (to remove a node attribute), establishing a clear semantic relationship. This relationship is not limited to functions within the same class. For instance, another function, `xml_attribute::set_value`, which sets the value of an attribute, is also semantically related to the above two functions. The prompt used to comprehend the purpose and the usage of API functions is illustrated in Figure 3 in Appendix B. With the advancements in LLMs for code summarization and functionality comprehension, such logical relationships can now be automatically extracted, allowing us to assess the semantic relevance among API functions. To compute this relevance, we structure a prompt around a key API function and several candidate API functions, instructing the LLM to determine which candidate functions have logical relationships with the key function based on their respective purposes. Let the key function be denoted as f_p, and one of the candidate functions be denoted as f_q. Their semantic relevance is then defined as:

$$sem\_rel(f_p, f_q) = \begin{cases} 1 & \text{if } f_q \text{ is chosen by LLM} \\ 0 & \text{if } f_q \text{ is not chosen by LLM} \end{cases} \tag{4}$$

3.2 Knowledge-Driven Generation

After acquiring the three types of knowledge, PromeFuzz generates harnesses based on the constructed knowledge base. In addition to the fundamental workflow (i.e., generating fuzzing harnesses and running them to detect crashes, as shown at the far-right of Figure 1), PromeFuzz incorporates three modules—*Scheduler*, *Generator*, and *Sanitizer* —and a feedback loop to optimize the generation and update the knowledge base.

3.2.1 Scheduler. In human-crafted fuzzing harnesses, APIs are organized and invoked in groups, similar to their usage in typical library consumer programs. Emulating these common API usage scenarios enhances harnesses to achieve improved performance and reduce false positives. Inspired by this, PromeFuzz selects a group of correlated target functions based on existing API call sequences or their scheduling scores, as detailed in Algorithm 1. The algorithm's first phase (lines 1-4) schedules existing call sequences, while the second phase (lines 5-16) schedules with computed scheduling scores.

Schedule with API Call Sequences. Developers often write tests and example programs for libraries, which typically contain API call sequences that are more likely to be constructed with comprehensive expert knowledge. These sequences are well-suited for reuse in harness generation. The *Scheduler* prioritizes these extracted sequences and sends them to the *Generator* module for harness generation until no unused sequences remain.

Schedule with Scheduling Score. After all existing call sequences have been utilized, the *Scheduler* will generate new function sets based on their correlation and generation-time coverage. As previously mentioned, API relevance is measured from three perspectives: type relevance, scope relevance, and semantic relevance. The overall relevance between functions f_p and f_q is then defined as the weighted average of these three metrics:

$$rel_{pq} = \frac{w_{typ} typ\_rel_{pq} + w_{scp} scp\_rel_{pq} + w_{sem} sem\_rel_{pq}}{w_{typ} + w_{scp} + w_{sem}} \tag{5}$$

where w_{typ}, w_{scp} and w_{sem} represent the weights for type relevance, scope relevance, and semantic relevance, respectively.

Meanwhile, the API coverage is fed back by the *Generator*. Each time a usable fuzzing harness is generated and stored, the *Generator* runs an AST-based processor to parse the API calls in the harness to get API coverage as well as to prevent hallucinations, which provides API invocation statistics to the *Scheduler*. The coverage of function f_p is calculated as:

$$cov_p = \frac{invoc(f_p)}{\sum_{f_i \in F} invoc(f_i)} \tag{6}$$

where $invoc(f_p)$ denotes the number of invocations of function f_p, and F represents the set of all API functions.

Subsequently, both API relevance and coverage are incorporated into the calculation to determine the scheduling score of an API function. For API coverage, the intuition is that the less an API is invoked, the higher its priority for scheduling. For API relevance, the relevance between the candidate function and all selected functions should be considered. Hence, the scheduling score of a candidate API f_p is defined as:

Algorithm 1: The algorithm to schedule API function sets

Input: API functions F, API call sequences S, The coverage of each function cov_x, The relevance between each two functions rel_{xy}, The maximum size of the function set $size_{max}$
Output: Scheduled API function set T

1 **if** $size_S \geq 1$ **then**
2 $T \leftarrow$ an arbitrary sequence from S;
3 $S \leftarrow S \setminus \{T\}$;
4 **return** T;
/* f_0 serves as a placeholder. The relevance of all APIs associated with it is 0. */
5 $T \leftarrow \{f_0\}$;
6 **while** $size_T \leq size_{max}$ **do**
7 $score_{max} \leftarrow -1$;
8 $t \leftarrow -1$;
9 **foreach** $f_i \in F \wedge f_i \notin T$ **do**
10 $score_i \leftarrow \frac{w_{cov}(1-cov_i)+w_{rel} \cdot MinMax(\sum_{f_j \in T} rel_{ij})}{w_{cov}+w_{rel}}$;
11 **if** $score_i > score_{max}$ **then**
12 $score_{max} \leftarrow score_i$;
13 $t \leftarrow i$;
14 $T \leftarrow T \cup \{f_t\}$;
15 $T \leftarrow T \setminus \{f_0\}$;
16 **return** T;

$$score_p = \frac{w_{cov}(1 - cov_p) + w_{rel} \cdot MinMax(\sum_{f_i \in T} rel_{pi})}{w_{cov} + w_{rel}} \quad (7)$$

where $MinMax(x) = \frac{x - \min(x)}{\max(x) - \min(x)}$ represents the Min-Max normalization, T denotes the set of already selected API functions, and w_{cov} and w_{rel} are the corresponding weight parameters. Given the scheduling scores of all candidate functions, the *Scheduler* operates in a greedy manner, iteratively selecting the API function with the highest score (lines 11~14) until a complete function set is formed.

3.2.2 Generator. With the aforementioned comprehended knowledge and scheduled API function set, the *Generator* collects the necessary knowledge based on the given target API function set for LLM-based code generation. The *Generator* directly retrieves API usage through API function names. The *Generator* also employs a graph traversal algorithm on the metadata dependency graph to get API dependent code. All reachable nodes in the graph are considered dependent language objects for the target function set, whose declarations are extracted and categorized, forming structured prompts for the LLM. Figure 4 in Appendix B presents a typical prompt for harness generation targeting the `pugixml` library.

3.2.3 Sanitizer. The fuzzing harnesses generated by the LLM may not function as intended. They can encounter errors, including compilation failures and runtime crashes.

Compilation Failures. For compilation failures during the build process, the *Sanitizer* directly prompts the LLM to apply fixes with the compilation error message, as these issues are typically caused by syntax violations and are relatively straightforward. Handling runtime crashes is more complicated. PromeFuzz addresses them using a crash analyzer and goes a step further by learning from these errors, feeding the acquired knowledge (i.e., API constraints) back into the knowledge base for continuous improvement.

Runtime Crashes. Specifically, runtime crashes can be categorized into two types: crashes caused by API misuse and real bugs in the library. Although a crash report is generated when a fuzzing harness is built with AddressSanitizer [43], the relevant information underlying the crash message is extensive and challenging to collect. To address this challenge, we propose the crash analyzer, which handles crashes in two ways: a quick path and a full-scale path.

If the crash occurs immediately after running the harness and is its first occurrence, it is considered an API misuse and is addressed through the quick path. The crash type and call stack are parsed from the crash message, which is then used to prompt the LLM to fix the fuzzing harness.

On the other hand, crashes that occur repeatedly or after a period of execution are meticulously addressed through the full-scale path. Firstly, the call stack is parsed to gather relevant information from the knowledge base, including the generated fuzzing harness, source code surrounding the call site, invoked API functions, and related data structures. A detailed crash report is then synthesized with this information. Advanced reasoning models, such as `DeepSeek-R1` and `OpenAI o1`, analyze the report to identify the root cause of the crash. If the crash is genuinely caused by a bug in the library, the crash report is recorded for further confirmation by researchers. Otherwise, it is classified as a false positive.

Figure 5 in Appendix B presents a prompt example for analyzing a crash within the `libtiff` library. As demonstrated, the LLM successfully identified the crash as a false positive based on the fuzzing harness code and the related function implementation. To prevent similar mistakes in the future, the reasoning model subsequently summarizes the correct usage of the relevant API functions, referred to as API constraints. If the LLM successfully fixes the fuzzing harness using these summarized API constraints, they are deemed valuable and incorporated into the API usage knowledge base, thereby enhancing future generation processes. For instance, as shown in Figure 7 in Appendix A, PromeFuzz analyzes the crash within `xml_attribute::set_value` and learns that the parameter must be null-terminated.

4 Implementation

We implemented PromeFuzz on top of the Clang/LLVM [23] and LangChain [4] frameworks, comprising approximately 9,000 lines of Python and 900 lines of C++ code, all of which are open-sourced. Key implementation details are outlined below.

C++ Language Support. Supporting C++ presents significantly greater challenges than C, thus previous works mainly support C, despite claiming compatibility with C++.

C++ programs often utilize advanced features for inheritance and polymorphism, posing unique challenges during our process. We addressed three major issues: 1) Unlike in C, C++ programs frequently employ function overloading, resulting in multiple functions sharing the same name. To handle this, we carefully distinguish these functions based on their source locations and explicitly prompt LLMs to differentiate between various overloads. 2) The initialization of C++ classes and structures typically relies on constructors. We identify constructors for each class, with particular attention to those of abstract classes, which can only be invoked through their derived classes. 3) Nested types, such as templates, introduce complex dependency relationships among other types. To address this, we decompose these complex types into basic types and filter out integrated types (e.g., `string`), enabling us to effectively extract dependencies. For remaining edge cases, the capabilities of LLMs help to bridge the gap, allowing valid fuzzing harnesses to still be generated successfully.

Metadata Extraction. Metadata extraction involves parsing language objects in library source code and identifying their attributes and dependencies. To ensure broad applicability, PromeFuzz uses Bear [40] to generate compile commands for each library and leverages a Clang-AST processor to build a language object database based on source code files. The language objects are categorized into four types: functions, classes, composite types, and typedefs. The declaration source code of these objects, along with surrounding comments, is also extracted. We then construct a metadata dependency graph using an algorithm that begins with API functions and recursively identifies dependent objects until no new dependencies remain. We define eight dependency relationships as follows:

- A function uses a composite type as a parameter or return value.
- A function belongs to a class.
- A composite type contains another as a member.
- A composite type is redefined via a typedef.

- A composite type belongs to a class.
- A class uses a function as its constructor.
- An abstract class is inherited by another class.
- A typedef defines a composite type.

In *Generator*, API-dependent code is extracted by traversing this graph, with a default depth limit of 3 to balance sufficient knowledge acquisition and computational efficiency.

API Functions Extraction. Another essential step is the extraction of API functions from the library. PromeFuzz first identifies all function objects declared in the header files of target library and then applies a filter process. Virtual functions, non-public functions, constructors, and destructors are filtered out. Additionally, PromeFuzz conducts a verification process, where a minimal invocation is attempted for each function. Functions that fail this test are also excluded. The remaining functions are classified as API functions.

Merging Fuzzing Harnesses. Due to LLM context limitations, only a few API functions can be tested within each generated fuzzing harness, resulting in a large number of individual harnesses. Executing them all introduces the challenge of efficiently allocating the fuzzing time budget. Inspired by prior work [2, 35], PromeFuzz employs an algorithm to merge all generated fuzzing harnesses into a single unified harness with one entry point. Within this merged harness, the specific test routine is selected based on the initial bits of the fuzzing input data. This approach not only simplifies execution but also enables coverage-guided harness selection, enhancing overall fuzzing efficiency.

Reducing LLM Cost. PromeFuzz employs a set of strategies to reduce expenses and enhance the affordability and practicality of our approach. Firstly, it structures knowledge efficiently by restricting metadata graph traversal depth to exclude unnecessary code. Declarations are unified under namespaces and classes for compactness. During multi-round fixing, the original harness generation code in the LLM's chat history is replaced with the latest response, reducing chat history length. Additionally, PromeFuzz selects LLMs based on task requirements. For summarization tasks in the comprehension phase, smaller models like `GPT-4o-mini` are used. For generative tasks in the generation phase, full-scale models like `DeepSeek-V3` are employed. Complex crash analysis tasks are handled by reasoning models like `DeepSeek-R1`. This alignment of LLMs with task demands improves system efficiency.

5 Evaluation

We evaluated PromeFuzz in real-world libraries and answered the following research questions (RQs):

RQ1. Is PromeFuzz effective in generating fuzzing harnesses?
RQ2. How effective is PromeFuzz in finding vulnerabilities?
RQ3. How does each component contribute to PromeFuzz's overall performance?

Experiment Setup. We conducted our experiments on a server equipped with two Intel® Xeon® Gold 6230R CPUs (52 cores each, 2.10 GHz), 128 GB of RAM, and Ubuntu 24.04.1 LTS (64-bit). We compared PromeFuzz with the state-of-the-art LLM-based automatic fuzzing harness generation tools, PromptFuzz [35] and CKGFuzzer [48], as well as the manually written fuzzing harnesses in OSS-Fuzz [42], if available for the target library. For multiple fuzzing harnesses generated for a library, we merged them into a single harness and performed fuzz testing on each merged harness using one CPU core for 24 hours. Each test was repeated 10 times to ensure the validity of the experimental results, following the recommendations of Klees et al. [21]. We utilized AFL++ [13] as the fuzzing engine and ensured consistency across experiments by using the same initial seed for fuzzing harnesses targeting the same library.

Comparison Tool. We compared PromeFuzz with PromptFuzz, CKGFuzzer and OSS-Fuzz-Gen, and manually crafted harnesses from OSS-Fuzz. For a fair comparison, we configured the underlying LLMs of both PromptFuzz and CKGFuzzer to use `DeepSeek-V3`. Harness generation quality is independent of the number of threads and is primarily determined by the LLM's capabilities and tool-specific stopping conditions. For CKGFuzzer and OSS-Fuzz-Gen, we adopted their default stopping conditions. For PromptFuzz, which lacks a defined stopping criterion, we terminated the generation phase once its LLM query cost matched that of PromeFuzz. In the fuzzing phase, each tool was given 24 CPU hours per project. Since CKGFuzzer failed to generate fuzzing harnesses for most libraries in the dataset, we instead used the pre-generated harnesses open-sourced by the authors. Additionally, the default configuration of OSS-Fuzz-Gen only cover a limited number of APIs. For a fair comparison, we extended its configuration to include all available APIs.

Dataset. We selected 22 real-world libraries from recent fuzzing research. Detailed information, including library versions, lines of code, number of APIs, and branch counts, is provided in Table 2. Of these, 18 are fully tested by OSS-Fuzz with manually written fuzzing harnesses. To ensure a fair comparison with PromptFuzz and CKGFuzzer, our dataset includes all 14 libraries used in their original evaluations. Additionally, we selected 5 C++ programs to evaluate the ability of PromptFuzz and CKGFuzzer, as their original datasets consist solely of C libraries.

5.1 RQ1: Generating Fuzzing Harnesses

We applied PromeFuzz to generate fuzzing harnesses for all 22 libraries in our dataset. During the experiments, we configured the *Comprehender* component to use the `GPT-4o mini` model [37] with 50 concurrent threads, the *Generator* component to use the `DeepSeek-V3` model [9] with 5 concurrent threads, and the *Sanitizer* component to use the `DeepSeek-R1` [16] model with 5 concurrent threads. The weights in Equation 5 are set as follows: $w_{typ} = 2$, $w_{scp} = 3$, and $w_{sem} = 5$, while the weights in Equation 7 are set to $w_{cov} = 3$ and $w_{rel} = 1$. The temperature parameter was set to 0.5 for all LLMs used in the experiment. We further discuss the model choice in Section 6.3 and the weight selection in Section 5.3.3.

The overall results of the experiment are presented in Table 2. PromeFuzz successfully generated fuzzing harnesses for 3,062 API functions (95.4% of the total number of API functions) across all 22 libraries. In total, PromeFuzz produced 2,577 fuzzing harnesses, of which 78.81% were valid and used for fuzzing. The entire process, including knowledge base construction and knowledge-driven generation, took 18 hours and 5 minutes, with a total cost of $15.89 and a consumption of 98.4 million tokens. On average, the time and cost per API were 21 seconds and $0.005, respectively. Although the *Comprehender* accounts for 80% of total query requests and token

Table 2: Overall results for PromeFuzz-generated fuzzing harnesses.

Tested Library					PromeFuzz's Cost			Coverage (#Branch / #API)				
Name	**Version**	**LoC**	**#API**	**#Branch**	**Time**	**Expense**	**Token**	**PromeFuzz**	**PromptFuzz**	**CKGFuzzer**	**OSS-Fuzz-Gen**	**OSS-Fuzz**
c-ares	7978cf7	44K	136	8,237	1h14m	$0.78	3.8M	**6,106** / 113	5,711 / 59	5,677 / 91	3,612 / 28	3,733 / 19
curl	4c50998	187K	96	29,942	31m	$0.28	1.4M	**7,347** / 95	5,006 / 87	6,143 / 91	759 / 27	950 / 7
lcms	04ace9c	46K	358	9,040	52m	$1.59	11.3M	**4,560** / 358	3,672 / 231	3,767 / 227	374 / 32	716 / 31
libjpeg-turbo	36ac5b8	64K	72	10,636	33m	$0.18	1.0M	4,274 / 71	3,520 / 78	N/A	957 / 14	**4,480** / 26
sqlite3	557974e	180K	270	50,903	1h13m	$2.44	17.1M	**28,225** / 246	26,263 / 160	N/A	17,987 / 17	25,568 / 15
libaom	99fcd81	530K	47	61,360	34m	$0.19	0.9M	**13,697** / 47	13,596 / 45	N/A	11,365 / 20	11,290 / 11
libpcap	2559282	42K	99	5,944	47m	$0.20	1.2M	**3,825** / 89	2,752 / 73	3,358 / 71	502 / 21	3,051 / 10
libvpx	9514ab6	370K	37	32,357	21m	$0.12	0.8M	**5,898** / 37	3,594 / 30	3,578 / 32	3,206 / 14	2,755 / 7
zlib	5a82f71	30K	89	2,720	2h06m	$0.17	0.9M	**2,430** / 89	2,099 / 85	2,343 / 66	1,865 / 34	1,789 / 29
re2	c84a140	28K	79	8,520	31m	$0.13	0.8M	**6,236** / 78	5,222 / 30	N/A	5,408 / 17	6,134 / 36
libmagic	dadc01f	18K	18	6,124	3m	$0.04	0.2M	**3,868** / 18	120 / 9	N/A	2,689 / 8	3,132 / 8
libpng	738f5e7	58K	256	6,999	51m	$1.10	7.6M	**3,538** / 251	3,296 / 228	N/A	881 / 16	2,081 / 27
ngiflib	db19270	1K	7	357	5m	$0.06	0.2M	**351** / 7	283 / 7	323 / 6	40 / 5	N/A
ffjpeg	caade60	2K	40	585	7m	$0.12	0.4M	**571** / 40	375 / 22	N/A	305 / 33	N/A
liblouis	3d95765	38K	30	6,247	10m	$0.07	0.3M	**4,704** / 30	2,220 / 17	N/A	1,723 / 22	3,797 / 5
libtiff	fcd4c86	92K	198	13,528	28m	$0.49	3.5M	**8,096** / 198	7,428 / 136	7,109 / 148	4,983 / 68	5,666 / 13
cjson	12c4bf1	17K	78	924	7m	$0.11	0.6M	**899** / 78	863 / 75	706 / 43	779 / 57	502 / 6
pugixml	06318b0	24K	221	5,957	1h24m	$0.32	2.0M	**3,957** / 221	N/A	N/A	369 / 2	3,385 / 7
tinygltf	a5e653e	193K	65	11,594	8m	$0.12	0.5M	**5,739** / 65	N/A	N/A	1,541 / 2	2,885 / 1
exiv2	04e1ea3	79K	880	44,320	5h46m	$7.17	43.4M	**13,580** / 816	N/A	N/A	8,144 / 7	9,857 / 17
loguru	4adaa18	3K	84	883	10m	$0.15	0.5M	**621** / 66	N/A	190 / 33	233 / 10	N/A
rapidcsv	083851d	6K	49	326	8m	$0.07	0.3M	**205** / 49	N/A	N/A	10 / 8	N/A
SUM			3,209	317,503	18h5m	$15.89	98.4M	**128,727** / 3,062	86,020 / 1,372	33,194 / 808	67,489 / 444	91,771 / 275

usage, it consumes only 10% of the total execution time. This efficiency is due to the independence of the tasks within *Comprehender*, enabling high concurrency and significantly reducing processing time. In contrast, increasing concurrency in the *Generator* can negatively affect API coverage and API constraint feedback.

We further compared the branch coverage of harnesses generated by PromeFuzz for libraries in the dataset against those produced by the baselines. The results are presented in Table 2, including the number of the covered branches (collected by llvm-cov [39]) and the directed invoked API functions in the harnesses. Specifically, the fuzzing harnesses generated by PromeFuzz achieve **1.50×** (128,727 / 86,020) higher branch coverage than those of PromptFuzz, **3.88×** (128,727 / 33,194) higher than CKGFuzzer, **1.91×** (128,727 / 67,489) higher than OSS-Fuzz-Gen, and **1.40×** (128,727 / 91,771) higher than OSS-Fuzz. If we exclude the libraries with `N/A` data from the comparison, the branch coverage of PromeFuzz is 1.22× (104,625 / 86,020) higher than PromptFuzz, 1.21× (40,133 / 33,194) higher than CKGFuzzer, 1.38× (126,979 / 91,771) higher than OSS-Fuzz. Table 7 in Appendix C presents the p-values from the Mann-Whitney U test comparing the branch coverage between PromeFuzz and the baselines. PromeFuzz significantly ($p < 0.05$) covers more branches in 94% (16 / 17) of libraries compared to PromptFuzz, 100% (10 / 10) compared to CKGFuzzer, and 94% (17 / 18) compared to OSS-Fuzz.

Our analysis reveals that a key reason for PromeFuzz's superior performance is its knowledge-driven approach, which enables more API function coverage. The fuzzing harnesses generated by PromeFuzz directly invoke 1.34× (1,845 / 1,372) more API functions than PromptFuzz, 1.40× (1,130 / 808) more than CKGFuzzer, 6.90× (3,062 / 444) more than OSS-Fuzz-Gen, and 10.5× (2,900 / 275) more than OSS-Fuzz. However, in the case of `libjpeg-turbo`, the manually crafted harnesses invoke only 26 API functions but still achieve higher branch coverage than automatically generated harnesses that invoke more API functions. This result demonstrates the advantage of domain-specific knowledge possessed by human experts.

The low branch coverage of PromptFuzz, CKGFuzzer and OSS-Fuzz-Gen is due to their lack of knowledge about library constraints. Consequently, these incorrectly generated harnesses crash early and are filtered out. For `pugixml`, OSS-Fuzz-Gen generated 152 harnesses but only 1 of them is valid for fuzzing. We discuss the details in Appendix D.

An additional reason for the poor performance of CKGFuzzer is its inability to properly handle LLM hallucinations. Taking `loguru` as an example, when CKGFuzzer queried LLM to generate harness for API function `get_verbosity_from_name`, the LLM hallucinated a function with the same signature instead of calling the actual library API function. This hallucination prevents the harness from covering intended code in the library. In contrast, PromeFuzz employs a Clang AST plugin to verify whether an API function is actually invoked, effectively eliminating LLM hallucinations.

We further analyze the reasons for the `N/A` data in Table 2. For OSS-Fuzz, the `N/A` indicates that the library does not contain manually crafted harnesses. This situation highlights the scarcity of fuzzing harnesses and emphasizes the need for automatic fuzzing harness generation. For PromptFuzz and CKGFuzzer, the `N/A` indicates a failure to generate fuzzing harnesses. PromptFuzz failed to generate harness for all C++ libraries in the dataset, because it cannot identify the function signature for C++ code. As discussed in Section 2.3, CKGFuzzer fails to properly utilize call graph knowledge and instead ports the entire call graph into the LLMs. Consequently, even for `libmagic` with only 18 API functions, CKGFuzzer was unable to generate fuzzing harnesses because the prompt exceeded

Table 3: Crashes found by PromeFuzz, PromptFuzz, CKG-Fuzzer, OSS-Fuzz-Gen and OSS-Fuzz.

Library	Prome		Prompt		CKG		OSS-Gen		OSS-Fuzz	
	#TP	#FP	#TP	#FP	#TP	#FP	#TP	#FP	#TP	#FP
c-ares	0	0	0	1	0	9	0	2	0	0
curl	0	0	0	1	0	8	0	0	0	0
lcms	0	1	0	2	0	13	0	1	0	0
libjpeg-turbo	0	1	0	4	-	-	0	2	0	0
sqlite3	0	0	0	10	-	-	0	1	0	0
libaom	0	0	0	6	-	-	0	2	0	0
libpcap	0	0	0	1	0	9	0	5	0	0
libvpx	0	0	0	0	0	3	0	1	0	0
zlib	0	0	0	3	0	7	0	1	0	0
re2	0	0	0	1	-	-	0	2	0	0
libmagic	2	0	0	0	-	-	0	0	0	0
libpng	0	0	0	6	-	-	0	0	0	0
ngiflib	2	0	0	6	2	0	0	1	-	-
ffjpeg	4	1	0	0	-	-	0	5	-	-
liblouis	12	0	1	5	-	-	1	1	5	0
libtiff	2	0	0	7	0	6	0	6	0	0
cjson	0	0	0	2	2	1	0	2	0	0
pugixml	0	0	-	-	-	-	0	0	0	0
tinygltf	0	0	-	-	-	-	0	0	0	0
exiv2	1	0	-	-	-	-	0	0	0	0
loguru	0	0	-	-	0	4	0	10	-	-
rapidcsv	3	0	-	-	-	-	0	0	-	-
SUM	26/22	3	1/1	55	4/3	60	1/1	42	5/5	0
Precision	**89.7%**		1.8%		6.3%		2.3%		100%	

The number of crashes confirmed by the developer is listed in the right of /.

Table 4: New real-world vulnerabilities found by PromeFuzz.

Library	ID	Vulnerability Type	Status	P	C	OG	OF
ngiflib	Issue 34	NULL Pointer Dereference	Fixed	✗	✓	✗	-
	Issue 36	NULL Pointer Dereference	Fixed	✗	✓	✗	-
ffjpeg	Issue 55	Use Uninitialized Variable	Reported	✗	-	✗	-
	Issue 57	Buffer Underflow	Reported	✗	-	✗	-
	Issue 58	Stack Buffer Overflow	Reported	✗	-	✗	-
	Issue 60	Stack Buffer Overflow	Reported	✗	-	✗	-
liblouis	Issue 1708	Invalid Free	Fixed	✗	-	✓	✗
	Issue 1709	Invalid Free	Fixed	✗	-	✗	✗
	Issue 1711	Memory Leak	Confirmed	✗	-	✗	✗
	Issue 1718_1	Memory Leak	Fixed	✗	-	✗	✗
	Issue 1718_2	Memory Leak	Fixed	✗	-	✗	✗
	Issue 1719	Heap Buffer Overflow	Fixed	✗	-	✗	✓
	Issue 1720	Heap Buffer Overflow	Fixed	✗	-	✗	✓
	Issue 1721	Heap Buffer Overflow	Fixed	✗	-	✗	✓
	Security j9qg	Infinite Loop	Confirmed	✗	-	✗	✗
	Security 3wwm	Heap Buffer Overflow	Confirmed	✗	-	✗	✓
	Security w63h	Heap Buffer Overflow	Confirmed	✗	-	✗	✗
libtiff	Issue 669	Reachable Assertion	Fixed	✗	✗	✗	✗
	CVE-2025-45701	Memory Leak	Fixed	✗	✗	✗	✗
exiv2	CVE-2025-26623	Use After Free	Fixed	-	-	✗	✗
libmagic	CVE-2025-55450	Integer Overflow	Fixed	✗	✗	✗	✗
	Bug tracker 640	Integer Overflow	Fixed	✗	✗	✗	✗
rapidcsv	Issue 186	Out-of-bounds Write	Fixed	-	-	✗	-
	Issue 187	Out-of-bounds Write	Fixed	-	-	✗	-
	Issue 188	Out-of-bounds Read	Fixed	-	-	✗	-

P: PromptFuzz **C**: CKGFuzzer **OG**: OSS-Fuzz-Gen **OF**: OSS-Fuzz

the maximum input length. In contrast, due to the optimizations detailed in Section 4, PromeFuzz consumed only around 3,000 tokens per query on average, well within the token limitation.

5.2 RQ2: Finding Vulnerabilities

In this section, we analyze the crashes detected by the fuzzing harnesses generated by PromeFuzz, PromptFuzz, CKGFuzzer, OSS-Fuzz-Gen, and manually crafted harnesses from OSS-Fuzz. For each crash triggered by the harness, we manually verified whether it was a bug in the library (True Positive, TP) or an API misuse (False Positive, FP). We also identified the root cause of crashes and clustered the crashes with the same root cause. Table 3 summarizes the experiment results. In addition to the number of true positives and false positives, we include the number of true positives confirmed by developers. PromeFuzz found 29 crashes with a precision of 89.7%, of which 26 are true positives, and 22 of them are confirmed by the developer (1 of them was reported by others). The high precision of PromeFuzz is due to the combination of the knowledge extracted by *Comprehender* and *Sanitizer*. The manually crafted fuzzing harnesses in OSS-Fuzz discovered only five crashes but achieved a precision of 100%. This is because the security experts who wrote these fuzzing harnesses possessed extensive expert knowledge. However, due to limited human resources, they cannot achieve full API coverage or craft a fuzzing harness for all libraries, thus they identified only a subset of crashes. In contrast, PromptFuzz, CKGFuzzer, OSS-Fuzz-Gen detected more crashes, identifying 56, 64 and 27 crashes, respectively. However, the majority of these crashes were false positives, resulting in low precision rates of only 1.8%, 6.3% and 2.3%.

Further analysis shows that PromeFuzz found 25 previously unknown real-world vulnerabilities. The details are listed in Table 4. For each vulnerability, we list its track ID, vulnerability type based on the Common Weakness Enumeration (CWE), and its current status in the table. Among the discovered new vulnerabilities, 21 of them are confirmed by the developer, 17 of them are fixed, and 3 of them are assigned CVE IDs. We also present whether the other works can find the new vulnerabilities.

Case Study. `Exiv2` is a C++ library for reading and modifying image file metadata. Although it has been extensively tested through OSS-Fuzz and no recent vulnerabilities have been reported, PromeFuzz successfully discovered a use-after-free vulnerability in `Exiv2`, which has been assigned CVE-2025-26623. We present the vulnerable code in Listing 2. Specifically, Lines 1~6 define the `TIFFSubIFD` class, which represents a sub-image file directory in a TIFF file. This class maintains an identifier for a new group, `newGroup_`, and a container of IFD objects, `ifds_`. Lines 8~9 define the copy constructor and lines 11~14 define the destructor. Lines 16~23 implement the `doAccept` method, which first calls a method on the visitor to handle the sub-IFD itself, then iterates over each contained IFD to pass them to the visitor for further processing. However, the copy constructor replicates only `newGroup_` but performs a shallow copy of `ifds_` (line 9), which means both the original and the copied `ifds_` end up sharing the same memory reference. When one of the objects deallocates this shared memory, the other is left with a

```
1  class TiffSubIfd : public TiffEntryBase {
2      friend class TiffReader;
3      // DATA
4      IfdId newGroup_;
5      Ifds ifds_;
6  };
7
8  TiffSubIfd::TiffSubIfd(const TiffSubIfd &rhs)
9      : TiffEntryBase(rhs), newGroup_(rhs.newGroup_) {}
10
11 TiffSubIfd::~TiffSubIfd() {
12     for (auto &&ifd : ifds_)
13         delete ifd;
14 }
15
16 void TiffSubIfd::doAccept(TiffVisitor &visitor) {
17   visitor.visitSubIfd(this);
18   for (auto &&ifd : ifds_) {
19     if (!visitor.go(TiffVisitor::geTraverse))
20       break;
21     ifd->accept(visitor); /* UAF happens here */
22   }
23 } // TiffSubIfd::doAccept
```

Listing 2: The vulnerable code of CVE-2025-26623 in exiv2.

dangling pointer, leading to a use-after-free vulnerability at Line 21.

This vulnerability was identified due to PromeFuzz 's advanced C++ support. Specifically, the API `Exiv2::TiffParser::encode`, which has seven complex parameters, was successfully handled through detailed metadata extraction (including type declarations and constructors) combined with documentation knowledge (parameter specifications), enabling the LLM to generate valid function calls. No human-written harness exists for this API. Competing tools like PromptFuzz and CKGFuzzer failed due to limited C++ support and insufficient knowledge, making our tool uniquely capable of discovering this bug.

5.3 RQ3: Ablation Study

In this experiment, we evaluated the contribution of each component in PromeFuzz to fuzzing harness generation by disabling the *Comprehender* (`w/o Comp`), *Scheduler* (`w/o Sched`), and *Sanitizer* (`w/o San`) individually and regenerating fuzzing harnesses for the libraries in the dataset in Section 5.3.1, analyzed the effectiveness of the crash analyzer in *Sanitizer* for filtering crashes in Section 5.3.2 and evaluated the weight selection in Section 5.3.3.

5.3.1 Per-component Effectiveness. Table 5 presents the number of directly invoked API functions, branch coverage, and the number of LLM queries. Additionally, Figure 8 in Appendix E illustrates the evolution of branch coverage over time during fuzzing for harnesses generated by PromeFuzz with and without each component.

The results demonstrate that each component contributes to the generation of high-quality fuzzing harnesses. The branch coverage of harnesses generated by PromeFuzz with all components enabled is **1.16**× that of `w/o Comp`, **1.26**× that of `w/o Sched`, and **1.35**× that of `w/o San`. Similarly, the number of API functions directly invoked is 1.01×, 1.14×, and 1.31× higher, respectively. According to Table 7 in Appendix C, PromeFuzz covers significantly more branches in 77% (17 / 22) of libraries compared to `w/o Comp`, 95% (21 / 22) compared to `w/o Sched`, and 91% (20 / 22) compared to `w/o San`. The precision of vulnerability detection drops to 29.4% without Comprehender (`w/o Comp`), 20.0% without Scheduler (`w/o Sched`), and 2.7% without Sanitizer (`w/o San`).

Among all components, the *Sanitizer* has the most significant impact on the quality of fuzzing harnesses, which aligns with our intuition. Without the *Sanitizer* to filter false positives, many harnesses fail to run correctly, resulting in reduced coverage. While the loss in coverage of `w/o Comp` is relatively minor, the absence of relevant knowledge requires more fixes during the sanitization phase, leading to an increased number of queries needed to generate the harness. Harnesses are generated using randomly generated API sequences in `w/o Sched`, which are often irrelevant and therefore incapable of producing high-quality harnesses.

It is worth noting that the branch coverage of `w/o Comp` for the `exiv2` is higher than PromeFuzz. Upon analysis, we found that the code in `exiv2` contains detailed comments, and the documentation largely overlaps with these comments. Regardless of whether the *Comprehender* is enabled, PromeFuzz's *Preprocessor* extracts the corresponding commented code as part of the function signature. This redundancy between the information understood by the *Comprehender* and that provided by the *Preprocessor* may affect the quality of harnesses generated by the LLM.

5.3.2 Crash Analyzer. We further analyzed the effectiveness of the crash analyzer in *Sanitizer*. During the experiment in Section 5.1, the *Sanitizer* mitigated 191 crashes through the quick path and 112 crashes through the full-scale path in total. Details of the bug types filtered by PromeFuzz are listed in Appendix F. The Constraints Learning step in feedback process introduces minimal overhead (3.6% of queries) but is critical for reducing repeated LLM errors. Though the quick path method may treat real bugs as false positives, we manually verified all filtered crashes and confirmed they were all caused by API misuse. It is worth noting that during manual verification, we classified a crash flagged as a false positive by the crash analyzer as a library bug and reported it to `libpng`. However, the developer later identified it as an API misuse. The generated harness seems to do all the initialization properly but actually missed one, which required domain-specific knowledge to identify. We manually inspected all crashes and found only 3 misreports by the Sanitizer. Two were due to LLMs assuming APIs perform boundary checks, which the original implementations intentionally omit. The third involved a low-level API not intended for direct use. The details are presented in Appendix G.

5.3.3 Weight Selection. To evaluate the impact of weight parameters in Equation 5 and Equation 7, we varied their values and conducted experiments on a subset dataset of 10 libraries. Table 6 presents the branch coverage achieved by the generated fuzzing harnesses under different weight settings. Although our default weight settings do not yield the highest coverage for every individual library, they consistently rank among the top-performing configurations overall. Experimental results show that, within a certain range, increasing w_{cov} effectively improves overall coverage. The lowest coverage was observed when the $w_{cov} : w_{rel}$ ratio was 1:3. As w_{cov} increased, coverage improved, peaking at a 3:1 ratio. However, further increasing the ratio to 5:1 resulted in a decline. This trend is intuitive: insufficient emphasis on w_{cov} causes PromeFuzz to repeatedly generate harnesses for a limited set of frequently used API sequences, limiting exploration. Conversely,

Table 5: Coverage and query times of PromeFuzz with and without each component.

Tested Library			PromeFuzz			w/o Comp			w/o Sched			w/o San		
Name	**API**	**Branch**	**API**	**Branch**	**Query**	**API**	**Branch**	**Query**	**API**	**Branch**	**Query**	**API**	**Branch**	**Query**
c-ares	136	8,237	113	**6,106**	532	94	5,669	631	80	4,793	215	40	5,107	136
curl	96	29,942	95	**7,347**	226	95	2,738	273	90	4,452	58	63	3,158	78
lcms	358	9,040	358	**4,560**	480	341	4,174	409	269	3,344	214	313	4,481	273
libjpeg-turbo	72	10,636	71	**4,274**	159	72	4,005	68	72	3,774	33	62	3,369	64
sqlite3	270	50,903	246	**28,225**	736	250	22,570	669	191	18,880	296	170	16,191	311
libaom	47	61,360	47	**13,697**	99	45	13,452	162	45	13,641	19	38	11,799	46
libpcap	99	5,944	89	**3,825**	115	91	3,758	112	92	3,655	50	83	3,684	100
libvpx	37	32,357	37	**5,898**	55	37	4,383	52	34	3,108	19	35	3,713	55
zlib	89	2,720	89	2,430	106	89	2,307	82	87	2,019	25	89	**2,436**	68
re2	79	8,520	78	**6,236**	86	77	6,205	174	67	6,036	38	69	5,559	59
libmagic	18	6,124	18	**3,868**	26	18	3,679	38	18	3,551	5	18	3,685	30
libpng	256	6,999	251	**3,538**	335	251	3,220	423	249	3,038	116	219	2,930	217
ngiflib	7	357	7	**351**	75	7	347	124	6	302	12	7	287	32
ffjpeg	40	585	40	**571**	139	39	562	181	35	479	40	29	504	53
liblouis	30	6,247	30	**4,704**	107	30	4,642	497	20	4,358	42	17	4,457	54
libtiff	198	13,528	198	**8,096**	486	194	5,714	361	173	6,168	148	185	5,206	221
cjson	78	924	78	**899**	85	78	890	78	78	810	31	77	882	94
pugixml	221	5,957	221	**3,957**	197	215	3,918	329	187	3,449	70	203	3,524	252
tinygltf	65	11,594	65	**5,739**	88	65	3,045	70	65	2,695	18	54	2,794	31
exiv2	880	44,320	816	13,580	1,792	812	**14,655**	1,458	736	12,691	721	495	11,335	457
loguru	84	883	66	**621**	185	69	526	311	42	497	102	15	68	64
rapidcsv	49	326	49	**205**	63	50	193	102	40	117	14	43	169	52
SUM	3,209	317,503	3,062	**128,727**	6,172	3,019	110,652	6,604	2,676	101,857	2,286	2,324	95,338	2,747

Table 6: Branch coverage of PromeFuzz under different weight settings. (Deeper color intensity indicates a higher number of covered branches)

Library	default	$w_{cov}\uparrow$	$w_{rel}\uparrow$	$w_{typ}\uparrow$	$w_{scp}\uparrow$	eq1	eq2
curl	7,347	8,032	5,376	8,325	8,548	8,195	5,916
lcms	4,560	4,492	4,570	4,902	4,443	4,418	4,800
sqlite3	28,225	17,702	18,739	21,237	20,667	23,706	21,846
libaom	13,697	21,060	12,690	14,335	15,966	19,724	14,406
libvpx	5,898	7,576	4,224	4,436	3,648	3,857	4,766
libpng	3,538	3,540	3,598	3,464	3,612	3,777	3,567
ffjpeg	571	559	521	555	504	567	575
liblouis	4,704	3,343	4,080	4,796	4,623	4,691	4,990
cjson	899	891	857	884	904	900	894
tinygltf	5,739	2,917	2,901	2,981	2,716	3,113	3,890
SUM	75,178	70,109	57,555	65,913	65,630	72,947	65,650

default denotes $w_{cov} : w_{rel} = 3:1$ and $w_{typ} : w_{scp} : w_{sem} = 2:3:5$
$w_{cov}\uparrow$ denotes $w_{cov} : w_{rel} = 5:1$ and $w_{typ} : w_{scp} : w_{sem} = 2:3:5$
$w_{rel}\uparrow$ denotes $w_{cov} : w_{rel} = 1:3$ and $w_{typ} : w_{scp} : w_{sem} = 2:3:5$
$w_{typ}\uparrow$ denotes $w_{cov} : w_{rel} = 3:1$ and $w_{typ} : w_{scp} : w_{sem} = 5:3:2$
$w_{scp}\uparrow$ denotes $w_{cov} : w_{rel} = 3:1$ and $w_{typ} : w_{scp} : w_{sem} = 2:5:3$
eq1 denotes $w_{cov} : w_{rel} = 1:1$ and $w_{typ} : w_{scp} : w_{sem} = 2:3:5$
eq2 denotes $w_{cov} : w_{rel} = 1:1$ and $w_{typ} : w_{scp} : w_{sem} = 1:1:1$

when w_{rel} is underweighted, PromeFuzz may select poorly correlated API sequences, leading to a higher crash rate and reduced effective coverage. To support different use scenarios, PromeFuzz offers configurable weight parameters, allowing users to adjust them based on specific requirements.

6 Discussion

6.1 False Positives

Unlike application fuzzing, library fuzzing requires constructing fuzzing harnesses to invoke API functions. Consequently, crashes discovered during library fuzzing may be false positives. Identifying whether a crash is a false positive demands in-depth knowledge of the target library, which increases the risk of security researchers mistaking false positives for true vulnerabilities. In addition to the example provided in Section 2.2, even crashes assigned CVE IDs can suffer from this. For instance, PromptFuzz generated a fuzzing harness for `TIFFReadRGBATileExt` in `libtiff` that triggered a segmentation fault, submitted a related issue, and successfully obtained a CVE ID (CVE-2023-52356). However, the developers of `libtiff` clarified that *this is just a misuse of `libtiff` API by using invalid parameters in a function call* [5]. PromeFuzz leverages its reasoning model-based crash analyzer in *Sanitizer* to filter out a significant number of false positives. The crash analyzer reduces the manual analysis workload and improves the accuracy of crash classification. As a result, PromeFuzz could identify this as an API misuse as demonstrated in Figure 5 in Appendix B.

6.2 Failure Cases and Limitation

We conducted an in-depth analysis of failure cases in fuzzing harness generation and identified two primary categories of root causes, both of which persist despite incorporating domain knowledge distilled by LLMs. These cases expose the inherent limitations of relying solely on current LLM capabilities.

[5] https://gitlab.com/libtiff/libtiff/-/issues/622

```
1 Slice<byte *> makeSlice(DataBuf &buf, size_t begin, size_t end);
2
3 Slice<const byte *> makeSlice(const DataBuf &buf, size_t begin,
4     size_t end);
5
6 template <typename T>
7 Slice<T> makeSlice(T &cont, size_t begin, size_t end);
8
9 template <typename T>
10 Slice<T *> makeSlice(T *ptr, size_t begin, size_t end);
11
12 template <typename container>
13 [[nodiscard]] Slice<container> makeSlice(container &cont);
```

Listing 3: The 5 function signatures of `Exiv2::makeSlice`.

Error-prone API functions. LLMs are prone to misusing certain API functions, often triggering runtime errors such as `free` mismatches or buffer read/write overflows. Some of these APIs exhibit strong semantic correlations and are frequently scheduled together in the same API sequence. As a result, harnesses involving these APIs may consistently fail and are eventually discarded by the scheduler. This issue can be mitigated by increasing the maximum number of fix attempts, but this may lead to significantly higher token consumption. To balance robustness and efficiency, we set the default maximum fix attempts to 5 while making it configurable for users.

Incorrect handling of C++ API overloads. LLMs often struggle to accurately distinguish between overloaded C++ functions. Even when provided with explicit function signatures and instructions, the model may generate harnesses for the wrong overloaded variant. This issue accounts for nearly all failure cases observed in the `exiv2` library. As shown in Listing 3, `Exiv2::makeSlice` has five overloaded variants with different parameters, yet the LLM fails to generate a harness for the specific variant used on line 3. These cases underscore the fundamental difficulty LLMs face in following instructions within complex contexts. This challenge might be mitigated by future advancements in LLM capabilities and targeted tooling optimizations.

6.3 Influence of Different LLM Models

We observed that the capabilities of the LLM significantly affect PromeFuzz's performance. We tested on part of the dataset, the experiment results show that using reasoning models (i.e., `DeepSeek-R1`) in *Generator* improves coverage by 3%∼10%, but significantly increases runtime by 10× and API expenses by 7×. Conversely, with local models like Qwen2.5-72B in *Comprehender* and *Generator*, hardware constraints (e.g., an A6000 GPU with 48GB VRAM) prevent concurrent execution, increasing runtime and reducing accuracy. Furthermore, during the experiments, we found that different task requirements favor specific model characteristics. The *Comprehender* component performs adequately with faster, cost-efficient models, while *Sanitizer* benefits from more accurate reasoning models despite their slower speed, as *Sanitizer* requires higher precision and is invoked less frequently.

6.4 Multi-Language Support

Supporting multiple languages has long been a significant challenge for existing automatic fuzzing harness generation [2, 5, 18, 31, 35, 48, 54, 57]. This difficulty primarily arises from the intricacies introduced by high-level language features, such as classes, templates, and virtual methods, which substantially increase the complexity of applying program analysis techniques. From the outset of our design, PromeFuzz has aimed to address this challenge by effectively combining program analysis with the capabilities of LLMs. Through these efforts, PromeFuzz implements support for both C and C++ languages and provides a foundation for straightforward extensions to other programming languages.

7 Related Work

7.1 Automatic Fuzzing Harness Generation

Existing methods for automatic fuzzing harness generation [2, 5, 15, 18–20, 27, 31, 49, 54, 55] primarily rely on program analysis techniques and existing consumer code. Based on the inferred API information, they reason about the dependency of internal functions and use that as guidance to generate the calling sequences with well-formatted arguments. For example, FUDGE [2] constructs the context for a fuzz driver by slicing the code of the target function. FuzzGen [18], APICraft [55], RULF [20], and RPG [49] analyze the dependencies for API functions to generate harnesses by merging different API call sequences. GraphFuzz [15], Hopper [5], and Rubick [54] modify the API sequence as well as the input seeds based on the data flow analysis, interpreter, or automaton. AFGen [31] generalizes harness generation to arbitrary functions, but it produces false positives that require manual filtering. NEXZZER [27] employs a dynamic graph structure, APIGraphs based on a consumer graph or type matching, and can be updated according to fuzzing feedback to record and infer relationships. Unfortunately, the aforementioned automated generation methods can only address the syntactic correctness of API invocations. They lack a deep semantic understanding of the target and are unable to effectively handle implicit constraints and the intricate relationships between API functions.

7.2 LLM for Fuzzing Harness Generation

LLMs can semantically understand code, allowing them to partially infer constraints and generate fuzzing harnesses that are both syntactically and semantically correct. PromptFuzz [35] and OSS-Fuzz-Gen [28] utilize basic information such as function signatures as input. CKGFuzzer [48] further improves harness quality by incorporating call graph knowledge. However, the above LLM-based works fail to effectively extract and utilize knowledge from the library, leading to fuzzing harnesses that cannot properly satisfy constraints. This results in false positives and limits the ability to explore the states of the target library.

7.3 LLM for Fuzzing

In addition to generating fuzzing harnesses, LLMs are also utilized in other processes of fuzzing [7, 12, 45, 51]. ChatFuzz [17] utilizes LLMs to mutate the seeds from the seed pool and generate format-conforming inputs for general applications. Though it performs well in programs with explicit syntax rules, its performance is less effective for programs with complex, non-trivial syntax without the corresponding knowledge. TitanFuzz [10] generates unit test code for deep learning libraries to identify bugs, while FuzzGPT [11]

enhances this capability by integrating historical bug knowledge derived from issues and pull requests. WhiteFox [50] and Fuzz4All [47] test multiple programming languages and features by generating diverse and realistic code snippets. ChatAFL [36] utilizes LLMs to generate message sequences, facilitating the discovery of bugs in protocol implementations.

8 Conclusion

In this paper, we presented PromeFuzz, a knowledge-driven approach for automated fuzzing harness generation. By systematically integrating structured knowledge from code metadata, documentation, and API correlations, PromeFuzz enables LLMs to generate harnesses that are both syntactically correct and semantically aware of implicit API constraints. Our end-to-end pipeline further enhances robustness through automated compilation fixes, real-time coverage feedback, and a dedicated sanitizer to filter false positives. Experimental results on 22 real-world projects demonstrate PromeFuzz 's superiority over state-of-the-art tools PromptFuzz, CKGFuzzer and OSS-Fuzz-Gen, as well as manually crafted fuzzing harnesses. It achieves 1.50× higher branch coverage than PromptFuzz, 3.88× higher than CKGFuzzer, 1.91× higher than OSS-Fuzz-Gen, and 1.40× higher than OSS-Fuzz, while discovering 25 previously unknown vulnerabilities (with 21 confirmed by the developer and 3 confirmed CVEs) at a precision of 89.7%.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and feedback. We also thank Yihua Ma for his assistance with various aspects of the experiments. This research was supported, in part, by National Natural Science Foundation of China (Grant No.62232016, 62472414), Research Project of Institute of Software Chinese Academy of Sciences (ISCAS-ZD-202402), Youth Innovation Promotion Association CAS, Ant Group Postdoctoral Programme and Ant Group. The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] 2022. CodeQL. https://codeql.github.com.
[2] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. 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*. 975–985.
[3] M. Böhme, V. Pham, and A. Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2016), 489–506.
[4] Harrison Chase. 2022. *LangChain*. https://github.com/langchain-ai/langchain
[5] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative Fuzzing for Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. 1600–1614.
[6] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.
[7] Yiran Cheng, Hong Jin Kang, Lwin Khin Shar, Chaopeng Dong, Zhiqiang Shi, Shichao Lv, and Limin Sun. 2025. Towards Reliable LLM-Driven Fuzz Testing: Vision and Road Ahead. *arXiv preprint arXiv:2503.00795* (2025).
[8] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 677–693.
[9] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437
[10] 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.
[11] 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.
[12] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.
[13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.
[14] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium*.
[15] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. In *2022 IEEE/ACM 44rd International Conference on Software Engineering (ICSE)*.
[16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).
[17] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. *arXiv preprint arXiv:2306.06782* (2023).
[18] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.
[19] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. UTopia: Automatic Generation of Fuzz Driver Using Unit Tests. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. 2676–2692.
[20] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust library fuzzing via API dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 581–592.
[21] 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*. 2123–2138.
[22] Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, and Nafise Eskandani. 2024. LLM-based and retrieval-augmented control code generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*. 22–29.
[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society.
[24] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*.
[25] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 474–499.
[26] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*.
[27] Jiayi Lin, Qingyu Zhang, Junzhe Li, Chenxin Sun, Hao Zhou, Changhua Luo, and Chenxiong Qian. 2025. Automatic Library Fuzzing through API Relation Evolvement. In *Proceedings of the 2025 Annual Network and Distributed System Security Symposium (NDSS), Virtual*.
[28] Dongge Liu, Oliver Chang, Jonathan metzman, Martin Sablotny, and Mihai Maruseac. 2024. *OSS-Fuzz-gen: Automated Fuzz Target Generation*. https://github.com/google/oss-fuzz-gen
[29] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.
[30] Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. 2023. VD-Guard: DMA Guided Fuzzing for Hypervisor Virtual Device. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1676–1687.
[31] 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)*. IEEE, 1901–1919.

[32] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. 2021. InstruGuard: find and fix instrumentation errors for coverage-based greybox fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 568–580.
[33] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung Won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *60th Annual Meeting of the Association for Computational Linguistics, ACL 2022*. Association for Computational Linguistics (ACL), 6227–6240.
[34] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Security Symposium*.
[35] 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*. 3793–3807.
[36] 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)*, Vol. 2024.
[37] OpenAI. 2024. GPT-4o mini: Advancing Cost-Efficient Intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence
[38] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *EMNLP (Findings)*.
[39] LLVM Project. 2025. llvm-cov - emit coverage information. https://llvm.org/docs/CommandGuide/llvm-cov.html Accessed: 2025-03-17.
[40] rizsotto. 2022. rizsotto/Bear: Bear is a tool that generates a compilation database for clang tooling. https://github.com/rizsotto/Bear
[41] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.
[42] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. (2017).
[43] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.
[44] Flavio Toffalini, Nicolas Badoux, Zurab Tsinadze, and Mathias Payer. 2025. Liberating Libraries through Automated Fuzz Driver Generation: Striking a Balance without Consumer Code. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2123–2145.
[45] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).
[46] 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. In *NDSS*.
[47] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[48] Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. 2025. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 243–254.
[49] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 2024. RPG: Rust Library Fuzzing with Pool-Based Fuzz Target Generation and Generic Support. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13.
[50] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 709–735.
[51] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* (2024), 100211.
[52] Yuanping Yu, Xiangkun Jia, Yuwei Liu, Yanhao Wang, Qian Sang, Chao Zhang, and Purui Su. 2022. HTFuzz: Heap Operation Sequence Sensitive Fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
[53] Michal Zalewski. 2013. American fuzzy lop (AFL) fuzzer. http://lcamtuf.coredump.cx/afl.
[54] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. 2023. Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation. In *USENIX Security Symposium*.
[55] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. 2811–2828.
[56] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
[57] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. IntelliGen: Automatic Driver Synthesis for Fuzz Testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 318–327.
[58] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. STOCHFUZZ: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 659–676.

A Workflow Examples for PromeFuzz

Figure 7 illustrates the workflow of PromeFuzz using examples from the `pugixml` library.

Preprocessor. The *Preprocessor* extracts the metadata dependency graph and the consumer call graph from the code base. In the dependency graph from Figure 7, `xml_node::remove_attribute` is a member of the `xml_node` class and takes an `xml_attribute` structure as its parameter, making it dependent on these two language objects as well as their respective constructors. In the call graph, `xml_node::remove_attribute` and several other APIs are invoked by the same test runner function, indicating their usage relationships. This information is then structured into the knowledge base, specifically as API dependent code and API correlation.

Comprehender. The *Comprehender* module extracts API implementations and vectorizes API documentation, integrating them into a unified database. LLMs then retrieve and summarize this information to synthesize API usage knowledge. The summarized usage knowledge for `xml_node::remove_attribute` is shown in Figure 7.

Scheduler & Generator. Based on the previously extracted knowledge, PromeFuzz schedules API sequences and prompts LLMs to generate fuzzing harnesses. Figure 7 shows a generated fuzzing harness for `xml_node::remove_attribute` and other correlated API functions.

Sanitizer. When crashes occur during the execution of fuzzing harnesses, *Sanitizer* analyzes the failures and attempts to learn from API misuse. In Figure 7, the LLM initially fails to recognize that the parameter of `xml_attribute::set_value` must be null-terminated, resulting in a crashing harness. By analyzing the crash, the LLM infers the correct invocation requirement and summarizes a new API constraint, thereby preventing similar errors in future generations.

B Prompt Examples for PromeFuzz

Figure 3 presents a prompt used by PromeFuzz when comprehending the usage of the `pcap_create` function for the `libpcap` library. Figure 4 shows a prompt for generating a fuzzing harness for the `pugixml` library. Figure 5 illustrates a prompt for analyzing a false positive crash within the `libtiff` library.

C P-values for Branch Coverage

Table 7 presents the p-values of the Mann-Whitney U test comparing the branch coverage of harnesses from PromeFuzz, the competing tools, and the ablation study.

D Analysis for low branch coverage of PromptFuzz

Listing 4 presents a fuzzing harness generated by PromptFuzz for `ngiflib`. Lines 3~12 define a structure `ngiflib_gif`. It contains a union `input` that can either store a file pointer `file` or a memory buffer `buffer`. It also includes the `mode` field, which is intended to track which input method is being used. The harness creates an in-memory file, validates the input data, tests the API function `LoadGif`, and cleans up resources in the end. However, `Ngiflib` is a small library containing only seven API functions and relatively obscure with fewer than 100 stars, limiting the knowledge of existing LLMs. As a result, without extra knowledge, LLMs do not recognize the constraints between `mode` and `input`. In our experiments, PromptFuzz generates 33 fuzzing harnesses that invoke `LoadGif`, all of which are similar to the Listing 4 and use only the file-based input, resulting in reduced code coverage. This is because PromptFuzz does not recognize that `mode` must be set to `NGIFLIB_MODE_FROM_MEM` when using a memory buffer.

E Branch Coverage in Ablation Study

Figure 8 presents the details of branch coverage in the ablation study.

F Bug Types Filtered by Crash Analyzer

The bug types are presented in Figure 6, with buffer overflows accounting for 42.90% and null pointer dereferences for 34.65%. The remaining bugs include memory leaks, use-after-free, double frees, mismatched frees, and floating-point exceptions.

TASK

You are required to identify a particular API function's purpose and u sage according to the source code and documentation. Consider:
- What is the API function's design aim?
- How should the function be used?
- How does it behave in various scenarios?

Don't add uncertain information. Be concise (under 300 characters).

CONTEXT

```
# Information about `pcap_create`
## Function Implementation
```c
pcap_t *pcap_create(const char *device, char *errbuf) {
...
## Documentation Excerpts
1. Excerpt from mandoc.txt:
```
NAME
    pcap_create - create a live capture handle
DESCRIPTION
    pcap_create() is used to create a packet capture handle to look at packets on the network. source is a string that specifies the network device to open; on all supported Linux systems, as well as on recent v ersions of macOS and Solaris, a source argument of "any" or NULL ca n be used to capture packets from all network interfaces. The latter s hould not be confused with all available capture devices as seen by p cap_findalldevs(3PCAP), which may also include D-Bus, USB etc.
errbuf is a buffer large enough to hold at least PCAP_ERRBUF_SIZE
...
```

RESPONSE

The `pcap_create` function is designed to create a live capture handl e for network packet capture. It initializes a `pcap_t` structure but do es not activate the capture; activation is done via `pcap_activate`. Th e `source` parameter specifies the network device (e.g., "any" or NUL L for all interfaces), and `errbuf` stores error messages, which holds at least PCAP_ERRBUF_SIZE characters.

Figure 3: The prompt for comprehending the usage of the `pcap_create`.

G False Positive from PromeFuzz

Listing 5 shows a simplified fuzzing harness for `lcms` generated by PromeFuzz that causes a false positive. The harness invokes the API `cmsDoTransform`, passing two memory buffers (`inputBuffer` and `outputBuffer`) and the corresponding length (lines 14~20). However, an overflow error occurs when the parameter `Size` equals the provided buffer length because the pixel size (lines 7~8) must be

TASK

Write a C++ fuzzing harness program to fuzz the library pugixml using the information provided below. Reply with only the content of the C++ source file, without any additional explanation. Follow:
1. Prepare the environment necessary before invoking the target function. Identify the required data types and classes, initialize them with appropriate values.
2. Conduct any required cleanup afterward. Pay attention to the return value of the function and handle exceptions properly.
3. Wrap all logic within the `extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)` function, which acts as the entry point for fuzzing.
...

CONTEXT

Target API Functions
You should invoke the functions in the following order: `pugi::xml_node::child` -> `pugi::xml_node::attribute` -> ...
Below is the detailed information:
1. `pugi::xpath_query::evaluate_string`
The `pugi::xpath_query::evaluate_string` function evaluates an XPath query as a string in a given XML context, performing type conversion if needed. It returns the result as a string, handling memory allocation errors with exceptions unless `PUGIXML_NO_EXCEPTIONS` is defined.
This function is a member of class pugi::xpath_query. Its signature is:
```cpp
string_t evaluate_string(const xpath_node& n) const;
size_t evaluate_string(char_t* buffer, size_t capacity, const xpath_node& n) const;
```
You should invoke all 2 overloads above for this function.
...
Related Data Types
```cpp
namespace pugi {
struct xml_node_struct {
...
## Related Constructor Methods
```cpp
namespace pugi {
class pugi::xml_node {
 explicit xml_node(xml_node_struct* p);
...
## Required Header Files
You need to include `pugixml.hpp` to invoke functions above.

**RESPONSE**

```cpp
#include "pugixml.hpp"
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
...
}
```

Figure 4: The prompt for generating fuzzing harness for `pugixml`.

TASK

As an expert developer familiar with the `libtiff` library, you encounter a crash when running fuzz test for it. Your task is to investigate the crash by answering the following questions:
- What kind of crash is it?
- Which part of the library is causing the crash?
- What is the root cause of the crash?
- Is the crash due to a bug in the library or misusing?
Below are the fuzzing harness that triggers the crash and its crash report, please analyze them and offer a concise explanation of it.

CONTEXT

Fuzzing harness
```c
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
...
  TIFFGetField(tif, TIFFTAG_IMAGEWIDTH, &width);
  TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &height);
  uint32_t *raster = (uint32_t *)malloc(width * height * sizeof(uint32_t));
  TIFFReadRGBATileExt(tif, col, row, raster, stop_flag);
... }
```
Crash report
Crash reason
AddressSanitizer: SEGV on unknown address 0x7fae47ef6070
Backtrace
The signal is caused by a WRITE memory access.
#0 TIFFReadRGBATileExt
libtiff/tif_getimage.c:3395:9
      memmove(raster + (size_t)(tile_ysize - i_row - 1) * tile_xsize,
...   ^
Related source code
```c
// Part of the `TIFFReadRGBATileExt` function:
  for (i_row = 0; i_row < read_ysize; i_row++) {
    memmove(raster + (size_t)(tile_ysize - i_row - 1) * tile_xsize,
...
## Related data definitions
...

**RESPONSE**

The crash is a segmentation fault (SEGV) caused by a WRITE memory access violation in `TIFFReadRGBATileExt` due to an out-of-bounds `memmove` operation on the `raster` buffer.
### Root Cause
The fuzzing harness allocates `raster` based on image dimensions (`TIFFTAG_IMAGEWIDTH`/`TIFFTAG_IMAGELENGTH`), but `TIFFReadRGBATileExt` uses **tile dimensions** for memory offsets. If tile dimensions exceed, the undersized `raster` buffer overflows.
### Misuse vs. Library Bug
This is an **API misuse**.The harness incorrectly assumes image dimensions suffice for tile operations, violating libtiff's requirement to size buffers for tile dimensions.Libtiff's implementation is correct but crashes due to the undersized buffer provided by the harness.

**Figure 5: The prompt for analyzing a crash within `libtiff`.**
```

Table 7: Mann-Whitney U test p-values for branch coverage across 10 runs.

Library	PromptFuzz	CKGFuzzer	OSS-Fuzz-Gen	OSS-Fuzz	w/o Comp	w/o Sched	w/o San
c-ares	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**
curl	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**
lcms	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**1.37E-02**
libjpeg-turbo	**9.77E-04**	N/A	**9.77E-04**	1.00E+00	**9.77E-04**	**9.77E-04**	**9.77E-04**
sqlite3	**9.77E-04**	N/A	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**
libaom	6.52E-01	N/A	**1.95E-03**	**2.93E-03**	2.16E-01	2.16E-01	**1.95E-03**
libpcap	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-03**	**9.77E-04**	**9.77E-04**
libvpx	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**1.95E-03**
zlib	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	6.77E-01
re2	**9.77E-04**	N/A	**9.77E-04**	**9.77E-04**	**2.73E-02**	**9.77E-04**	**9.77E-04**
libmagic	**9.77E-04**	N/A	**9.77E-04**	**9.77E-04**	1.16E-01	**9.77E-03**	1.16E-01
libpng	**9.77E-04**	N/A	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**
ngiflib	**9.77E-04**	**9.77E-04**	**9.77E-04**	N/A	**2.54E-02**	**9.77E-04**	**9.77E-04**
ffjpeg	**9.77E-04**	N/A	**9.77E-04**	N/A	6.25E-02	**9.77E-04**	**9.77E-04**
liblouis	**9.77E-04**	N/A	**9.77E-04**	**9.77E-04**	2.16E-01	**6.84E-03**	**1.37E-02**
libtiff	**1.86E-02**	**1.95E-03**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**
cjson	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**
pugixml	N/A	N/A	**9.77E-04**	**9.77E-04**	**4.88E-03**	**9.77E-04**	**9.77E-04**
tinygltf	N/A	N/A	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**	**9.77E-04**
exiv2	N/A	N/A	**9.77E-04**	**9.77E-04**	9.99E-01	**1.95E-03**	**9.77E-04**
loguru	N/A	**9.77E-04**	6.25E-02	N/A	**9.77E-04**	**9.77E-04**	**9.77E-04**
rapidcsv	N/A	N/A	**9.77E-04**	N/A	**9.77E-04**	**9.77E-04**	**9.77E-04**
P <0.05	16	10	21	17	17	21	20

```c
#define NGIFLIB_MODE_FROM_FILE  (0x00)
#define NGIFLIB_MODE_FROM_MEM   (0x02)
struct ngiflib_gif {
    union {
        FILE * file;
        struct {
            const u8 * bytes;
            unsigned long count;
        } buffer;
    } input;
    u8 mode; // NGIFLIB_MODE_FROM_XXX, related to input union
};

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Step 1: Create a memory file handle from the input data
    FILE *in_file = fmemopen((void *)data, size, "rb");
    if (!in_file)
        return 0;
    // Step 2: Check if the data is a valid GIF by CheckGif
    u8 gif_buffer[4];
    fread(gif_buffer, sizeof(u8), sizeof(gif_buffer), in_file);
    if (CheckGif(gif_buffer) == 0) {
        fclose(in_file);
        return 0;
    }
    // Step 3: Load GIF from memory
    struct ngiflib_gif gif;
    memset(&gif, 0, sizeof(gif)); // 0, NGIFLIB_MODE_FROM_FILE
    gif.input.file = in_file; // Set the input to the file
    if (LoadGif(&gif) != 0) {
        fclose(in_file);
        return 0;
    }
    // Step 4: Cleanup resources
    GifDestroy(&gif);
    fclose(in_file);
    return 0;
}
```

Listing 4: A fuzzing harness from PromptFuzz for `ngiflib`.

accounted for in the calculation. If the documentation does not explicitly specify this constraint, the analysis LLM may misinterpret it as the library's responsibility to handle, incorrectly identifying it as a bug in the library. Notably, the LLM correctly identified this as an API misuse in most cases, as evidenced by our repeated tests (10 trials) which consistently produced accurate results. While prior

```c
void CMSEXPORT cmsDoTransform(cmsHTRANSFORM  Transform,
    const void* InputBuffer,
    void* OutputBuffer,
    cmsUInt32Number Size) {
    _cmsTRANSFORM* p = (_cmsTRANSFORM*) Transform;
    cmsStride stride;
    stride.BytesPerPlaneIn = Size * PixelSize(p->InputFormat);
    stride.BytesPerPlaneOut = Size * PixelSize(p->OutputFormat);
    p->xform(p, InputBuffer, OutputBuffer, Size, 1, &stride);
}

int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    // ...
    uint8_t inputBuffer[1024];
    uint8_t outputBuffer[1024];
    memset(inputBuffer, 0, sizeof(inputBuffer));
    memcpy(inputBuffer, Data, Size < sizeof(inputBuffer) ?
            Size : sizeof(inputBuffer));
    cmsDoTransform(transform, inputBuffer,
            outputBuffer, sizeof(inputBuffer));
    // ...
    return 0;
}
```

Listing 5: The simplified fuzzing harness from PromeFuzz for `cmsDoTransform` in `lcms` that causing a false positive.

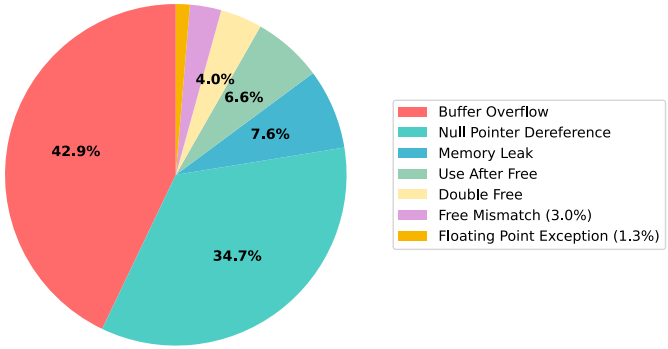


Figure 6: Filtered bug types.

work [25] proposed a voting system to mitigate model uncertainty through aggregated query results, we leave this approach as future work due to both the infrequency of such errors in our experiments (3 out of 303 cases) and the significant time and cost overhead of repeated queries.

H Comparison with Conventional Approaches

We also compared PromeFuzz with conventional approaches, Hopper [5] and libErator [44], in terms of branch coverage. For a fair comparison, we used the intersection of the evaluated libraries as the benchmark dataset. Table 8 and Table 9 present the results of the branch coverage comparison. PromeFuzz outperforms both tools across all evaluated libraries, achieving 51.13% higher branch coverage than Hopper and 78.36% higher than libErator. This improvement reaffirms the superiority of LLM-based methods over conventional approaches.

I Influence of Documentation

During harness generation, 20.5% of APIs had associated documentation. Documented APIs achieved a success rate of 97.27%,

compared to 94.94% for undocumented ones. This indicates that combining code analysis with learned knowledge helps reduce reliance on explicit documentation and improves overall robustness. We also investigated the impact of misleading documentation and found that, although the initially generated harness might be incorrect, the *Sanitizer* module can correct it through iterative fixing and learning. However, misleading documentation may increase the cost of generation. We leave the verification of documentation correctness as future work.

Listing 6 shows an incorrect fuzzing harness generated for the `ngiflib` library due to misleading documentation by PromeFuzz. Specifically, the documentation falsely indicates that `0x0` is a valid memory mode for `struct ngiflib_gif`. Consequently, PromeFuzz produces a harness similar to that generated by PromptFuzz, as shown in Listing 4. However, with the *Sanitizer* module enabled, PromeFuzz collects the crash report and the implementation of the crashing function (e.g., lines 6–12 in Listing 6), and feeds this information back to the LLM to correct the harness. Given this additional context, the LLM correctly distinguishes between memory mode and file mode, and regenerates a valid fuzzing harness. This example demonstrates that *Sanitizer* can eliminate the impact of misleading documentation through iterative LLM queries.

Table 8: Branches comparison with Hopper.

Library	PromeFuzz	Hopper	Δ%
c-ares	6106	4912	**+24.31%**
lcms	4560	3361	**+35.68%**
sqlite3	28225	13953	**+102.29%**
libaom	13697	9744	**+40.57%**
libpcap	3825	3012	**+26.99%**
libvpx	5898	3278	**+79.93%**
zlib	2430	1813	**+34.05%**
re2	6236	5656	**+10.27%**
libmagic	3868	2420	**+59.86%**
libpng	3538	3537	**+0.04%**
cjson	899	777	**+15.72%**
SUM	79283	52461	**+51.13%**

Table 9: Branches comparison with libErator.

Library	PromeFuzz	libErator	Δ%
c-ares	6106	4554	**+34.07%**
libaom	13697	7222	**+89.65%**
libpcap	3825	2261	**+69.23%**
libvpx	5898	2880	**+104.81%**
zlib	2430	1600	**+51.85%**
libtiff	8096	3747	**+116.05%**
cjson	899	696	**+29.11%**
SUM	40951	22960	**+78.36%**

```
1  /// Misleading Documentation: When call `LoadGif`,
2  /// init union `gif.input.file` when `gif.mode` is `0x0`
3  /// init union `gif.input.buffer` when `gif.mode` is `0x2`
4
5  /* Called by API LoadGif */
6  static u8 GetByte(struct ngiflib_gif * g) {
7      if(g->mode & NGIFLIB_MODE_FROM_MEM) {
8          return *(g->input.buffer.bytes++);
9      } else {
10         return (u8)(getc(g->input.file));
11     }
12 }
13
14 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
15     struct ngiflib_gif gif;
16     memset(&gif, 0, sizeof(gif));  // NGIFLIB_MODE_FROM_FILE
17     gif.input.buffer.bytes = data;
18     gif.input.buffer.count = size;
19     LoadGif(&gif);
20     // ...
21     return 0;
22 }
```

Listing 6: A wrong fuzzing harness for ngiflib generated by PromeFuzz with misleading documentation.

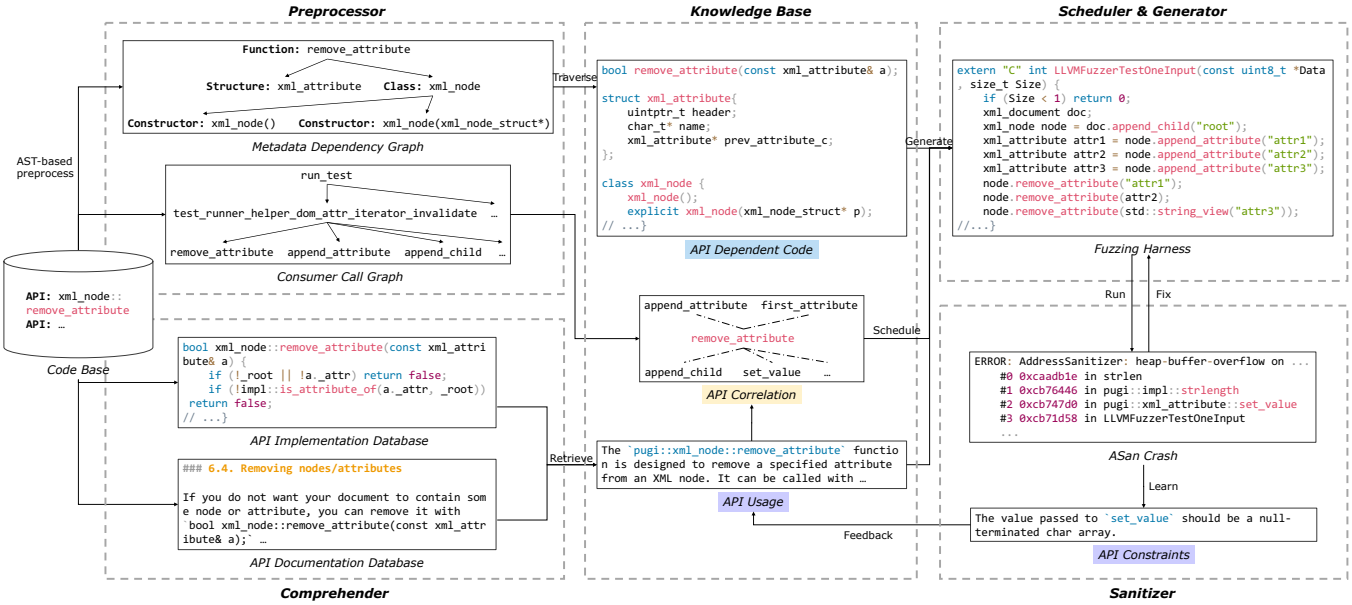


Figure 7: Workflow of PromeFuzz with examples.

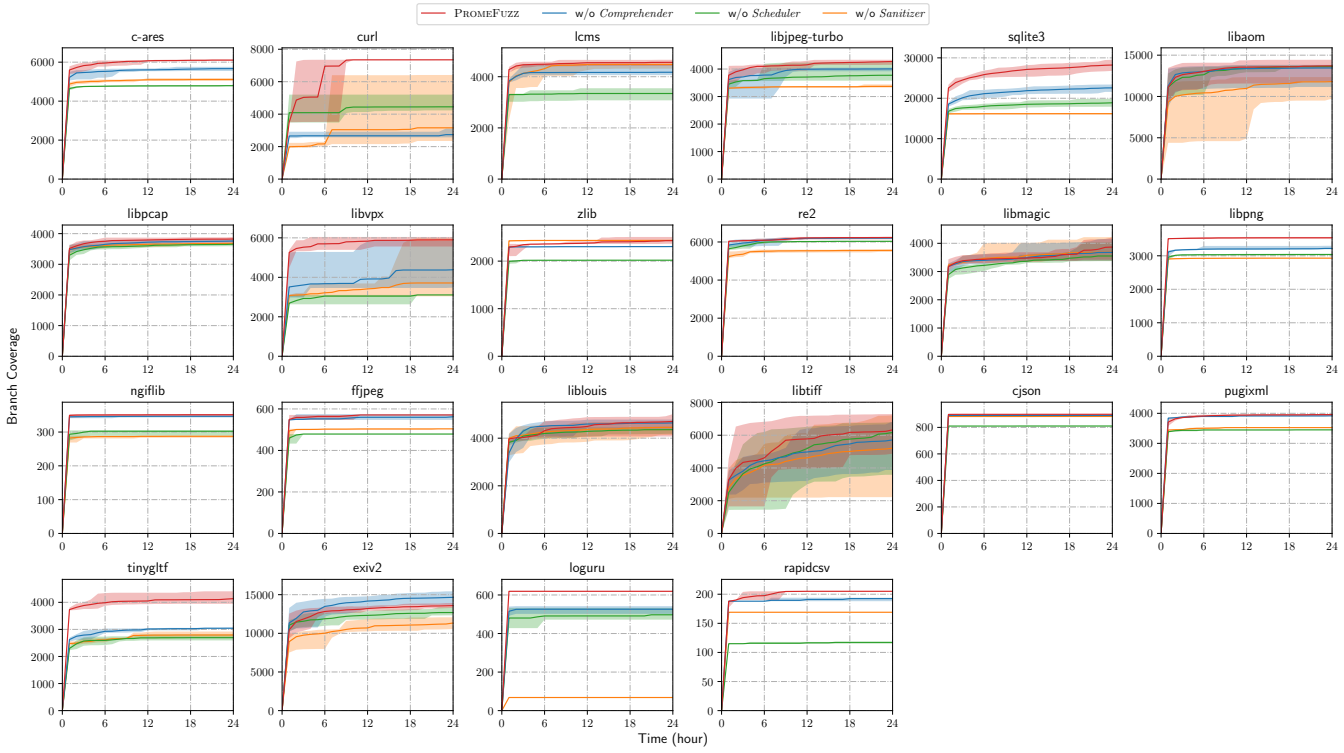


Figure 8: Branch coverage of PromeFuzz with and without each component.