Effective Command-line Interface Fuzzing with Path-Aware Large Language Model Orchestration

Momoko Shiraishi
The University of Tokyo
shiraishi@os.is.s.u-tokyo.ac.jp

Yinzhi Cao
Johns Hopkins University
yinzhi.cao@jhu.edu

Takahiro Shinagawa
The University of Tokyo
shina@is.s.u-tokyo.ac.jp

***Abstract*—Command-line interface (CLI) fuzzing tests programs by mutating both command-line options and input file contents, thus enabling discovery of vulnerabilities that only manifest under specific option-input combinations. Prior works of CLI fuzzing face the challenges of generating semantics-rich option strings and input files, which cannot reach deeply embedded target functions. This often leads to a misdetection of such a deep vulnerability using existing CLI fuzzing techniques.**

In this paper, we design a novel Path-guided, Iterative LLM-Orchestrated Testing framework, called PILOT, to fuzz CLI applications. The key insight is to provide potential call paths to target functions as context to LLM so that it can better generate CLI option strings and input files. Then, PILOT iteratively repeats the process, and provides reached functions as additional context so that target functions are reached.

Our evaluation on real-world CLI applications demonstrates that PILOT achieves higher coverage than state-of-the-art fuzzing approaches and discovers 51 zero-day vulnerabilities. We responsibly disclosed all the vulnerabilities to their developers and so far 41 have been confirmed by their developers with 33 being fixed and three assigned CVE identifiers.

1. Introduction

Fuzzing is a well-known and widely adopted approach to systematically discover software bugs and vulnerabilities, e.g., in Command-Line Interface (CLI) programs, with unexpected, malformed, or random inputs. Traditional fuzzing techniques, according to a survey of 102 recent fuzzing papers [1], often operate with fixed option configurations, making them challenging to reach deeply-embedded vulnerable functions that are only triggerable with certain option strings. Even recent directed fuzzing [2], [3], which use distance metrics to guide mutations toward target functions, may face similar challenges because without a correct option combination, distance metric may stay at infinity regardless of other mutations.

Therefore, an important problem that CLI fuzzing approaches need to address is the generation of different combinations of option strings. Traditional CLI fuzzing approaches [4], [5], [6], [1] rely on CLI option parsers, which only capture syntactic knowledge but not semantic relationships between different options. ProphetFuzz [7], the only, state-of-the-art LLM-based CLI fuzzing approach, parses the man pages of CLI options using LLM to generate CLI option combinations that are likely to trigger vulnerabilities. However, sometimes, man pages do not contain the deep knowledge of different option combination either.

More importantly, one unaddressed research challenge—beyond the aforementioned option string combination—is the generation of input files. Specifically, input files need to satisfy certain structural properties, which include but are not limited to proper file headers, valid codec parameters, correct container formats, and well-formed data structures. However, to the best of our knowledge, such input files are mostly provided beforehand as prior knowledge with manual efforts in state-of-the-art approaches. This not only brings additional human efforts, but also leads to challenges in triggering certain parts of CLI tools, which require a specific format.

In this paper, we design a novel Path-guided, Iterative LLM-Orchestrated Testing (PILOT) framework to fuzz CLI applications. The key idea is that PILOT identifies potential call paths to target functions and incorporates these paths into prompts as context so that PILOT can have a better understanding of the command line specification, thus generating different combinations of CLI option strings. Meanwhile, PILOT automatically discovers and orchestrates native command-line tools to generate semantically valid input files that enable reaching the target functions. The entire process is iterative. If the inputs from LLMs cannot reach the target function, PILOT provides feedback indicating which functions along the candidate paths are covered and then re-queries LLM for another set of potential option combinations and input files.

We implemented a prototype of PILOT and plan to release it at https://github.com/momo-trip/PILOT-code. Our evaluation across 43 widely used programs shows that PILOT discovered 51 zero-day vulnerabilities. We responsibly disclosed all of them to their developers and so far, 41 have been confirmed by developers with 33 being fixed and three assigned CVE identifiers. We also compared PILOT with state-of-the-art CLI fuzzing approaches, and the results showed substantial improvements in code coverage, i.e., 16.0% over CarpetFuzz [4], 36.2% over ProphetFuzz [7], 21.8% over ZigZagFuzz [1], and 57.9% over SelectFuzz [3].

2. Overview

In this section, we first describe a motivating example of seed generation for CLI fuzzing, and then give an overview of our solution.

2.1. A Motivating Example

We describe a motivating example with a zero-day vulnerability found by PILOT in FFmpeg [8], a multimedia framework for video and audio processing, which is widely used in popular applications like web browsers, media servers, and video conferencing systems. The vulnerability is a zero-day buffer overflow due to a negative size parameter, located in FFmpeg's Real-time Transport Protocol (RTP) streaming functionality of mp4 files [9]. It can be triggered either locally via command line or remotely through network streaming protocols, leading to arbitrary code execution through controlled heap/stack corruption. We have responsibly disclosed this vulnerability to the FFmpeg developers, who have acknowledged and fixed the vulnerability. CVE assignment is in progress.

Figure 1 shows a simplified version of the vulnerable code. Line 1 shows the command line that triggers the vulnerability. Specifically, the sample size table (stsz) atom in the crafted MP4 file ("malicious.mp4") in a valid H.264 format specifies a sample size with trailing byte for exploitation. The root cause of the vulnerability is that FFmpeg does not properly validate Network Abstraction Layer (NAL) unit sizes when processing H.264/HEVC streams for RTP transmission, leading to a negative size parameter passed to `memcpy` at Line 44. Specifically, the vulnerability occurs when the stsz, i.e., each frame's byte size, declares a frame size with 1-3 trailing bytes beyond the complete NAL unit. After a complete NAL unit is processed, 1 trailing byte, for example, remains and the loop at Line 22 continues. At Line 25, `ff_nal_mp4_find_startcode()` returns NULL since `end - r < 4`. The code sets `r1 = end` at Line 26, then executes `r += 4` at Line 27, causing `r = end + 3`. The calculation `len = r1 - r = -3` at Line 32 produces a negative value passed to `nal_send()` without validation. At Line 44, `memcpy(s->buf_ptr, buf, -3)` casts -3 to `size_t`, becoming `0xFFFFFFFFFFFFFFFD` (18 exabytes). Given a typical 1500-byte RTP buffer, this attempts to copy 12 trillion times more data than the buffer can hold, resulting in a massive buffer overflow.

```
1  // exploit code: ./ffmpeg -i malicious.mp4 -c copy -f rtp
       rtp://127.0.0.1:1234
2  // ffmpeg.c - Main entry point
3  int main(int argc, char **argv) {
4     // ... initialization ...
5     ret = transcode();  // Start transcoding
6     // the call chain is transcode -> muxer_thread ->
           av_interleaved_write_frame -> rtp_write_packet ->
           ff_rtp_send_h264_hevc -> nal_send
7     return ret;
8  }
9  // rtpenc.c - RTP encoder
10 static int rtp_write_packet(AVFormatContext *s1, AVPacket
       *pkt) {
11    // ... RTP packet setup ...
12    if (st->codecpar->codec_id == AV_CODEC_ID_H264 ||
13        st->codecpar->codec_id == AV_CODEC_ID_HEVC) {
14       ff_rtp_send_h264_hevc(s1, pkt->data, pkt->size);
             // Process H.264/HEVC
15    }
16    return 0;
17 }
18 // rtpenc_h264_hevc.c - H.264/HEVC RTP packetizer
19 void ff_rtp_send_h264_hevc(AVFormatContext *s1, const
       uint8_t *buf1, int size) {
20    const uint8_t *r = buf1, *end = buf1 + size;
21    RTPMuxContext *s = s1->priv_data;
22    while (r < end) {
23       const uint8_t *r1;
24       if (s->nal_length_size) {  // MP4 format
25          r1 = ff_nal_mp4_find_startcode(r, end,
                 s->nal_length_size);
26          if (!r1) r1 = end;
27          r += s->nal_length_size;  // VULNERABLE: No
                 boundary check
28       } else {
29          while (r < end && *r == 0) r++;
30          r1 = ff_nal_find_startcode(r, end);
31       }
32       int len = r1 - r;  // VULNERABLE: Can be negative
33       nal_send(s1, r, len, r1 == end);
34       r = r1;
35    }
36 }
37 static void nal_send(AVFormatContext *s1, const uint8_t
       *buf, int size, int last) {
38    RTPMuxContext *s = s1->priv_data;
39    // No validation of size parameter!
40    av_log(s1, AV_LOG_DEBUG, "Sending NAL %x of len %d\n",
41           buf[0] & 0x1F, size);
42    if (size <= s->max_payload_size) {
43       // ...
44       memcpy(s->buf_ptr, buf, size);  // CRASH: size = -3
45       // Cast to size_t = 0xFFFFFFFFFFFFFFFD
46       s->buf_ptr += size;
47    }
48 }
```

Figure 1: A motivating example illustrating a zero-day buffer overflow vulnerability found by PILOT. The code is slightly changed and simplified for better presentation.

2.2. Research Challenge and Overall Solution

The main research challenge is how to reach the vulnerable location with the correct inputs that can trigger the vulnerability. There are two conditions for this specific motivating example in Fig. 1: (i) a valid input file, i.e., a H.264/HEVC input file with proper NAL unit structures, and (ii) a correct combination of option strings, which are "-i file -c copy -f rtp address" for this vulnerability. More specifically, the input file must be valid in terms of containing valid H.264/HEVC codec identification to pass the condition `st->codecpar->codec_id == AV_CODEC_ID_H264` (at Line 12) and having properly initialized codec parameters to survive `av_interleaved_write_frame` processing.

To the best of our knowledge, state-of-the-art CLI fuzzing approaches [4], [7], [1] have difficulties in triggering this vulnerability, because they cannot generate the seeds that satisfy or are close to the aforementioned two conditions.

Valid Input File Generation. First, prior works mainly adopt a manually-crafted input file or one from the test cases, which is labor-intensive and may not capture the complex condition, e.g., satisfying the H.264/HEVC codec identification.

Semantic Option Combination. Second, they mainly adopt some semantic understanding methods, e.g, analyzing the source code's option parser (such as `getopt`), man pages (like `man ffmpeg`), or options from help output (such as `./ffmpeg --help`). None of them defines the behaviors of different option combinations like "-i file -c copy -f rtp address", thus failing to generate such option strings.

PILOT solves the problem to satisfy the two conditions by directed fuzzing using path-guided Large Language Model (LLM) prompting. Let us use `ffmpeg` as an example for illustration. First, PILOT queries an LLM to identify and install `ffmpeg` and generate base input files with proper codec parameters and container structures (e.g., via `ffmpeg -f lavfi -i testsrc -c:v libx264 output.mp4`). Note that the tool used for file generation may coincide with the fuzzing target, as in this example, but is generally a separate utility. This approach ensures deep semantic validity—files generated by actual encoders contain valid codec parameters and NAL unit structures, where subsequent mutations can introduce targeted malformations to trigger vulnerabilities. Second, PILOT leverages path-guided exploration and provides the LLM with the call chain from main to potential targets. This path-guided prompting enables the LLM to systematically examine code at each step, discovering that: (1) `print_sdp` is triggered by the `-sdp_file` option, (2) SDP generation works in conjunction with RTP output requiring `-f rtp`, and (3) HEVC codec input is needed to reach the HEVC-specific path through `extradata2psets_hevc`.

Note that the process is an iterative approach. PILOT executes each generated command, measures coverage to verify target function reachability, and provides feedback for iterative refinement when commands fail or the target function is not reached. By doing so, PILOT successfully generates validated initial seeds that reach deep program functionality. Again, we use the FFmpeg case as an example. After PILOT targets the high-centrality function `ff_isom_write_hvcc()` to explore RTP-related subsystems, PILOT further discovers the `nal_send()` function, which contains the vulnerability in out motivating example.

3. Design

In this section, we describe the design of PILOT from its system architecture and then present the detailed core techniques of PILOT.

3.1. Design Overview

PILOT leverages LLMs to generate effective initial seeds for CLI fuzzing, consisting of command-line strings and input files, which are later used by state-of-the-art CLI fuzzers. To achieve this, PILOT performs two key functions. First, it coordinates seed generation by iteratively communicating with the LLM, sending tailored prompts together with supplemental information such as the target function, potential call path to the function, and coverage feedback from the previous seed. Second, it provides an API that allows the LLM to interact with a sandboxed shell environment, where it can read target source-code fragments, write and execute a shell script, and install necessary command packages to generate input files. These capabilities enable the LLM to flexibly generate files and obtain contextual information as needed, facilitating iterative refinement of command-line seeds.

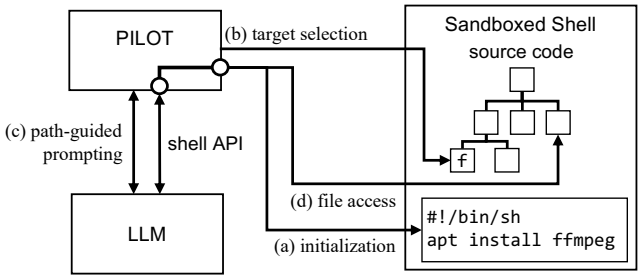

Figure 2: The design overview of PILOT.

Fig. 2 illustrates the overall design of PILOT. First, PILOT instructs the LLM to initialize the sandboxed shell environment for input file generation (Fig. 2 (a), Section 3.2). The LLM generates a script that installs the necessary command packages for input file generation, which is essential for producing valid and diverse input files. Next, PILOT selects a set of target functions (Fig. 2 (b), Section 3.3). Because LLM inference is costly, PILOT prioritizes target functions based on centrality metrics to achieve high coverage efficiently.

PILOT then prompts the LLM to generate command-line options intended to reach the selected target function (Fig. 2 (c), Section 3.4). To facilitate this, potential call paths are precomputed through static source-code analysis and provided to the LLM as contextual information to help it understand how the target function can be reached within the program. The LLM reads relevant source-code fragments as needed (Fig. 2 (d)) and returns a shell script to PILOT that includes candidate command-line strings and commands to generate input files designed to reach the target function.

PILOT executes the generated commands, collects coverage data, and, if the target function is not reached, prompts the LLM to refine the command line using the obtained coverage feedback. This iterative seed generation process is repeated up to a fixed number of attempts, and if the LLM still fails to produce a command line that reaches the target function, PILOT moves on to the next target function to improve overall coverage efficiently. The following subsections describe the details of the core techniques of PILOT in accordance with the procedure outlined in Algorithm 1.

3.2. Initialization for Input File Generation

Generating semantically valid input files is essential for effective CLI fuzzing. Rather than relying on random byte mutations that often fail parsing checks, PILOT auto-

matically identifies and configures command-line tools that produce well-formed input files (⓪ of Algorithm 1).

PILOT queries the LLM to discover standard command-line tools for creating valid inputs for a target command. To prevent trivial or ad-hoc file generation (e.g., `printf "\x89PNG..." > image.png` or `echo "data" > video.mp4`), PILOT employs carefully designed prompts that explicitly require well-established, standard command-line tools. For example, when fuzzing `ffmpeg`, PILOT asks the LLM to identify standard command-line tools that can produce valid file formats, such as `ffmpeg` itself, `convert` (ImageMagick) or `gstreamer`. The LLM then provides installation commands for the target OS. After executing the installation commands, PILOT verifies that each tool is correctly installed and accessible by checking command availability (using `command -v`). If a tool fails verification, PILOT repeats the installation and verification process. This iterative approach continues until the tool is successfully installed and verified.

3.3. Target Function Selection

While PILOT can potentially target any function in a program given appropriate prompts and context, practical constraints necessitate strategic selection of which functions to prioritize. Each seed generation attempt consumes LLM tokens, and the token budget is finite. Therefore, PILOT's primary objective is to maximize code coverage within this given token budget; PILOT must carefully select which functions to target to achieve the broadest coverage most efficiently.

Different programs exhibit distinct structural characteristics in their function call graphs. These structural properties can predict which functions will yield better coverage when targeted. Therefore, PILOT leverages graph centrality metrics, which identify structurally important functions in the call graph, to guide target selection. Specifically, PILOT employs an *adaptive target selection mechanism*. PILOT analyzes each program's call graph structure (Algorithm 1 ①) and applies learned decision rules (detailed in Section 5.4) to automatically select the optimal centrality-based strategy for that program (Algorithm 1 ②) . This adaptive approach maximizes coverage within the available token budget by selecting the strategy most suited to each program's unique structural characteristics.

3.4. Path-guided Context Prompting

Path-guided context prompting enhances LLM-based seed generation by providing explicit execution path information and enabling LLMs to understand the relationship between command-line arguments and target functions efficiently. This technique is complemented by an LLM-user interface that works together to overcome the inherent limitations of LLM context windows [10]. The following subsection describes the path-guided context prompting and the interface that enables autonomous code exploration.

Algorithm 1 Procedure of PILOT

Require: Program source code, set of functions $F = \{f_1, f_2, ..., f_n\}$
Ensure: Set of commands
NativeToolConfiguration($command$) {⓪ Environment setup}
Construct call graph $G = (V, E)$ where V is the set of functions and E represents function calls
$U \leftarrow F$ {U: Set of unvisited functions, initially all functions}
$N_{trial} \leftarrow$ Maximum number of attempts per function
$N_{target} \leftarrow$ Maximum number of functions to target
$\mathbf{c} \leftarrow$ Centrality score vector {① Initial function scoring}
while $U \neq \emptyset$ **and** $n_{processed} \leq N_{target}$ **do**
  $f_{target} \leftarrow \arg\max_{f_i \in U} \mathbf{c}[i]$ {② Select function with highest centrality score}
  $trial \leftarrow 0$ {Initialize attempts counter}
  covered $\leftarrow$ false
  while $trial \leq N_{trial}$ **and** not covered **do**
    $command \leftarrow$ GenerateCommand(f_{target}, G) {③ LLM command generation}
    $program\_state \leftarrow$ ExecuteCommand($command$) {④ Command execution}
    $cov \leftarrow$ AnalyzeCoverage($program\_state$) {⑤ Coverage analysis}
    if $f_{target} \in cov.covered\_functions$ **then**
      covered $\leftarrow$ true
      $U \leftarrow U \setminus \{f_{target}\}$ {Remove covered function from unvisited set}
    else
      $trial \leftarrow trial + 1$ {Increment attempts counter}
      if $trial > N_{trial}$ **then**
        $U \leftarrow U \setminus \{f_{target}\}$ {Switch to next target after max attempts}
      else
        $command \leftarrow$ RefineCommand($command, cov, f_{target}$) {⑥ Refine the command based on the feedback}
      end if
    end if
  end while
  $n_{processed} \leftarrow n_{processed} + 1$
  $G \leftarrow$ UpdateCallGraph(G, cov) {⑦ Coverage analysis}
  $\mathbf{c} \leftarrow$ Recalculate updated centrality scores {① Function scoring}
end while
$commands \leftarrow$ All generated commands
$seeds \leftarrow$ ConvertToSeeds($commands$) {⑧ Transform commands into fuzzing initial seeds}
$fuzzing\_results \leftarrow$ ExecuteFuzzing($seeds$) {⑨ Run fuzzing with generated initial seeds}
return $fuzzing\_results$

3.4.1. Initial Prompt

The initial phase leverages static analysis to identify potential execution paths from the main function to target functions. Fig. 3 shows an example prompt for exploring a target function (`ff_isom_write_hvcc()`) in the `ffmpeg` command. In the "Function Call Relationship" section (❶ of Fig. 3), multiple candidate execution paths from the main function to the target function are provided, each with their corresponding file locations and line numbers. The file name and line number for each function represent where the function is defined, not where it is called from, in order to provide stable reference points and reduce complexity, as multiple invocation paths may exist for the same execution sequence. For example, `main@ffmpeg.c:2932` indicates that the `main` function is located in the file `ffmpeg.c` at line `2932`. The LLM is then instructed to generate a shell script (`run_test.sh`) containing commands designed to

Example Prompt for {target command}

Task: Write ffmpeg test commands that reach the target function [ff_isom_write_hvcc@ffmpeg.c:1084].
Please include commands to create valid input files using appropriate standard tools: **ffmpeg, convert, sox, mencoder**. Do not manually construct file headers or use placeholder data. Invalid files cause early rejection and low coverage.

Target Function Definition:
int ff_isom_write_hvcc(AVIOContext *pb, ...) ...

❶ **Function Call Relationship from main() to the target function:**
- Path candidate 1
main@ffmpeg.c:2932
→ transcode/ffmpeg.c:2932
→ transcode_init@ffmpeg.c:2770
→ init_output_stream@ffmpeg.c:2770
...
→ extradata2psets_hevc@sdp.c:226
→ ff_isom_write_hvcc@hevc.c:1084

- Path candidate 2 ...
...

❷ **Response Format:** Please respond using one of the following three modes:
1) read_data: Request to read file contents
2) modify_data: Update file contents
3) execute_command: Run any command for testing

❸ **Directory Structure:**
workspace/
{target_program}-xxxx/
src/ ❹ # Source files (file1.c, file2.c, ...)
run_test.sh ❺ # LLM response script
...
execute.sh ❻ # Script for execute_command mode

Figure 3: Example prompt for exploring a target function. The prompt directs the generation of a shell script file (❺ `run_test.sh`) that will trigger execution to reach the target function.

trigger these execution sequences and reach the target function (Algorithm 1 ③). Due to the large number of potential execution paths—sometimes approaching 100—LLM input token limitations occasionally prevent including all possible paths at once. To overcome this constraint, PILOT exports the complete set of execution paths to an external file. This allows the LLM to access the full range of candidates through another read request-response chat (detailed in the subsequent Section 3.4.2), ensuring comprehensive analysis despite token restrictions.

3.4.2. Autonomous Code Exploration

Due to the limitations of LLMs' context windows, providing an entire codebase at once is often impractical. Instead, PILOT implements an autonomous code exploration method that enables LLMs to independently navigate and select relevant files for coding tasks. To enable this autonomous exploration, PILOT defines an API with the following three modes, requiring the LLM to select one in each response (described in ❷ of Fig. 3).

Read files. The `read_data` mode serves as the LLM's primary information-gathering mechanism. It allows the model to request specific file contents by providing the hierarchical target program directory structure (❸ of Fig. 3). This serves as a *map* of the project and provides the interface to know the details of each source file (❹). For larger files that exceed context window limitations, PILOT supports targeted examination of specific line ranges. The LLM can proactively specify particular sections of interest, or PILOT automatically provides guidance when token limits are reached. When the requested range exceeds the token limit, PILOT displays guidance in a prompt, such as "Exceeding token limit, content truncated. To view the complete content of [file_path], please use read_data mode and set file_slice (specified range) to read each section separately." This interactive capability enables the LLM to progressively build understanding of the project structure and functionality through strategic file selection and guided exploration.

Write files. The `modify_data` mode enables precise code modifications. The LLM can specify exact line ranges within files and provide replacement content, implementing targeted changes without needing to process entire files. This mode supports various modification patterns including complete replacements, deletions, and full file rewrites, giving the LLM fine-grained control over code changes.

Execute commands. The `execute_command` mode completes the interaction loop by allowing the LLM to run shell commands through a shell script in the workspace (❻ execute.sh) to test modifications or gather additional runtime information. By executing the script, the LLM can install necessary additional packages or extract information that might be challenging to parse from static source files.

3.5. Efficient Iterative Refinement

PILOT employs efficient iterative refinement techniques to maximize code coverage when LLMs struggle to reach target functions. This technique consists of two key components: covered-path feedback for learning from execution results and target switching for optimal resource allocation.

3.5.1. Covered-path Feedback

If the target function is not reached (Algorithm 1 ④ and ⑤), the feedback phase utilizes execution results to refine and guide subsequent seed generation attempts (Algorithm 1 ⑥). After executing generated commands, PILOT analyzes which functions were actually covered during execution. This feedback of executed function list enables the LLM to understand what went wrong and adjust its seed generation strategy accordingly to reach the target function.

3.5.2. Target Switching

While targeting high-centrality functions is an efficient method to increase coverage, in practice, LLMs cannot always cover target functions successfully. To efficiently allocate tokens when dealing with difficult-to-cover functions,

PILOT implements a pragmatic approach: the LLM generates commands for each target function up to a configurable parameter N_{trial}. If coverage remains unachieved after these attempts, PILOT redirects resources by switching to the next highest-priority function in the queue (Algorithm 1 ⑦).

Note that PILOT implements switching based on the number of test attempts rather than time limits. This design choice stems from PILOT's three distinct modes for request and response (described in Section 3.4.2). The inclusion of a read file mode prevents premature transitions to subsequent target generation before completing the seed generation attempt. Therefore, the limit is set at the point where the writing test operation consistently succeeds for N_{trial} consecutive runs.

3.5.3. Covering Branches

After successfully reaching a target function, PILOT refines test generation to achieve finer-grained coverage at the branch level. While function-level coverage provides a high-level view of program exploration, branch coverage reveals which execution paths within functions have been exercised. Once a function is covered, PILOT systematically guides the LLM by providing detailed context about uncovered branches. Specifically, PILOT includes in the prompt: (1) precise information about which branches remain uncovered, extracted through coverage analysis, (2) the target function definition, and (3) the directory structure to enable the LLM to comprehensively understand the codebase context. By explicitly supplying this structured information to the LLM, PILOT enables the model to reason about how to trigger uncovered branches through adjustments to command-line arguments, input file contents, or environmental conditions. This refinement process is iterative: PILOT repeats the generation-execution-analysis cycle until at least one new branch is discovered, or until a maximum of two refinement attempts have been made.

3.6. Initial Seed Extraction from LLM-generated Test Suites

The iterative exploration cycle described above produces multiple shell scripts (`run_test1.sh`, `run_test2.sh`, etc.) that execute the target program with different configurations. To use these tests as fuzzer seeds, PILOT transforms them into the format required by the target fuzzer by extracting command-line option strings and collecting input files (Algorithm 1 ⑧). PILOT is designed to generate initial seeds that can be utilized by various state-of-the-art CLI mutators, including CarpetFuzz, ZigZagFuzz, and SelectFuzz.

Extracting command-line options. Different mutators require seeds in specific formats. For example, CarpetFuzz [4] expects a single-line format: `{command} [options] -i @@`, where `@@` is replaced by the fuzzer with mutated input files. Converting complex shell scripts into this format requires semantic understanding of the script structure. For example, consider the following shell script generated by PILOT:

TABLE 1: POWER dataset for baseline comparison.

Program	LoC	# Files	# Functions	Package name
avconv [11]	680,276	1,193	11,074	libav-git-c464278
bison [12]	530,528	133	1,497	bison-3.7.6
cflow [13]	80,174	28	400	cflow-1.6
cjpeg [14]	112,689	39	312	libjpeg-turbo-2.1.0
dwarfdump [15]	183,051	109	1,503	libdwarf-20210528
ffmpeg [16]	1,611,669	2,040	20,448	ffmpeg-N-103440
gm [17]	587,458	161	1,727	GraphicsMagick-1.3.36
gs [18]	3,527,477	1,497	21,673	ghostpdl-9.54.0
mpg123 [19]	113,461	70	733	mpg123-1.28.2
nasm [20]	158,205	79	832	nasm-2.15.05
objdump [21]	4,628,717	136	2,977	binutils-2.36.1
pspp [22]	732,585	435	4,765	pspp-1.4.1
readelf [21]	4,628,717	130	1,289	binutils-2.36.1
tiff2pdf [23]	173,153	34	846	tiff-4.3.0
tiff2ps [23]	173,153	38	867	tiff-4.3.0
vim [24]	918,746	135	6,864	vim-8.2.3113
xmllint [25]	549,947	45	3,070	libxml2-2.9.12
xmlwf [26]	67,906	10	417	expat-2.4.1
yara [27]	62,743	64	603	yara-4.1.1

```
#!/bin/bash
# Commands for preparing input files
dd if=/dev/zero of=raw.yuv bs=1 count=12288
ffmpeg -f rawvideo -pix_fmt yuv420p -s 64x64 -r 1\
  -i raw.yuv -c:v libx265 -preset ultrafast \
  -x265-params keyint=1:aud=1 -frames:v 1
      input.mp4

# The main test command
timeout 3s ./ffmpeg -re -i input.mp4 -c:v copy \
  -f rtp rtp://127.0.0.1:5030
```

PILOT uses the LLM to extract the essential command that reaches the target function and format it as a fuzzer seed, `ffmpeg -re -c:v copy -f rtp rtp://127.0.0.1:5030 -i @@`. The LLM identifies the relevant program invocation, removes auxiliary commands (e.g., `timeout`), and reformats the command-line arguments into the fuzzer's expected format. This semantic extraction enables PILOT to handle diverse shell script structures automatically.

Collecting input files. To obtain the input files referenced in each test, PILOT executes all generated shell scripts (`run_test1.sh`, `run_test2.sh`, etc.) once and collects their outputs. For the example above, executing the script produces `input.mp4`, which becomes the corresponding seed file.

4. Implementation and Experimental Setup

This section describes the implementation and experimental setup.

Implementation. Our implementation contains 12,872 Lines of Python Code. The default LLM is based on Claude Sonnet 4.0 (claude-sonnet-4-20250514) [28]. GPT-4.1 (gpt-4.1) [29] is also employed for the comparison. PILOT's uses libclang [30] parser to generate call graphs, which is a C interface to the Clang compiler and provides access to Abstract Syntax Tree (AST) of the target C code. It also

uses the NetworkX (nx) [31] library for graph analysis and centrality calculations.

Setup. We conducted our experiments on Microsoft Azure virtual machines. Seed generation was performed on a Standard_L8as_v3 instance with 8 vCPUs, 64 GB RAM, and 512 GB storage running Ubuntu 22.04 LTS. Fuzzing campaigns were executed on a Standard_F32s_v2 instance with 32 vCPUs, 64 GB RAM, and 512 GB storage running Ubuntu 20.04 LTS. We set the maximum number of iterative refinements per function (N_{trial}) to 2 and the number of target functions (N_{target}) to 10. To reduce probabilistic behavior, the temperature [32] is set to 0, and `max_tokens` parameter [32], which indicates the maximum output tokens, is set to 4,096. All target programs were compiled with AddressSanitizer [33] enabled to detect memory safety violations including buffer overflows, use-after-free, and NULL pointer dereferences.

Dataset. We selected 43 C programs in total for our evaluation, consisting of three groups:

(i) POWER Dataset (20 programs). We first included the 20 programs from the POWER [34] dataset, which has been widely used for command-line fuzzing evaluation in prior work including CarpetFuzz [4], ProphetFuzz [7], and ZigZagFuzz [1]. These programs represent diverse applications ranging from parsers to multimedia tools and network utilities, with codebases between 62,743 and over 1,000,000 lines of code.

(ii) CarpetFuzz Dataset (13 programs). To enable comprehensive comparison with CarpetFuzz, we included the 13 additional programs that CarpetFuzz and ProphetFuzz evaluated.

(iii) Extended Dataset (10 programs). To evaluate PILOT's ability to discover zero-day vulnerabilities in actively maintained software, we selected 10 additional open-source projects based on two criteria: (1) high community adoption (GitHub stars $> 1,000$) and (2) recent developer activity (commits within the past 6 months). This extended evaluation is enabled by PILOT's key advantage: unlike CarpetFuzz, which relies solely on publicly available initial seeds and thus cannot evaluate arbitrary programs, PILOT generates seeds automatically for any target program.

For groups (i) and (ii), we used the same older versions as prior work to enable fair performance comparison. However, to discover zero-day vulnerabilities, we also evaluated these 33 programs at their latest versions. Combined with the 10 programs in group (iii), we performed zero-day vulnerability analysis on all 43 programs at their latest versions. Table 1 summarizes the 20 POWER dataset programs, while the remaining 23 programs are detailed in the artifact.

Baselines. In the evaluation, we conduct comprehensive experiments by integrating it with four state-of-the-art fuzzing frameworks.

- CarpetFuzz [4]: A state-of-the-art non-LLM approach that provides both initial seed generation and mutation capabilities. It leverages natural language processing to analyze command-line option inter-relationships from documentation for seed generation, and employs grammar-based mutations for fuzzing.
- ProphetFuzz [7]: A state-of-the-art LLM-based approach for initial seed generation that leverages LLM's knowledge on high-risk option combinations.
- ZigZagFuzz [1]: A state-of-the-art CLI fuzzing mutator that variates both option configurations and input files.
- SelectFuzz [3]: A state-of-the-art directed fuzzing approach that selectively guides exploration toward specific target locations by identifying and prioritizing relevant code paths.

TABLE 2: Comparison of initial seed generation methods and mutation strategies.

Approach	Initial seed generation	Mutation
CarpetFuzz [4]	CarpetFuzz	CarpetFuzz
ProphetFuzz [7]	ProphetFuzz	CarpetFuzz
PILOT-CF	PILOT	CarpetFuzz
ZigZagFuzz [1]	(Default) dictionary	ZigZagFuzz
PILOT-ZZ	PILOT	ZigZagFuzz
SelectFuzz [3]	No initial seeds	SelectFuzz
PILOT-SF	PILOT	SelectFuzz

Table 2 summarizes our evaluation setup. We integrate PILOT's seed generation with each mutation method to evaluate its effectiveness across different mutation strategies. Each fuzzing setup was run for 24 hours with 5 independent trials per program, and we measure the average coverage across these runs. For CarpetFuzz mutator, we compare PILOT-CF against two baseline seed generation approaches: the native CarpetFuzz method and ProphetFuzz. ProphetFuzz builds upon CarpetFuzz's mutator as its default base and provides a new initial seed generation method. For ZigZagFuzz mutator, we compare PILOT-ZZ against a standard dictionary-based approach (ZigZagFuzz). For SelectFuzz, which originally operates without initial seeds, we evaluate whether providing seeds generated by PILOT (i.e., PILOT-SF) can improve performance over the vanilla SelectFuzz.

To enable CLI fuzzing across all approaches, we apply the following modifications. For ZigZagFuzz, while it can leverage built-in initial seed options of target programs, these options are program-specific and not universally applicable. We therefore prepare a default dictionary by extracting options from program help screens. For input files, we use the same default seed files from the fuzzing framework (AFL++'s basic seed corpus). For SelectFuzz, which by default lacks an interface for mutating option strings, we integrate AFL++'s argv fuzzing interface [35] into its fuzzer component.

Preliminary Experiment. To analyze function call graph properties with respect to coverage performance and establish an effective function prioritization strategy, we conducted a preliminary experiment on the 20 POWER dataset programs. For each program, we selected 10 target functions

TABLE 3: Target selection strategies evaluated in our experiments.

Strategy	Metric	Description
CLOSE	Closeness centrality	Measures average distance from a function to all other functions in the call graph
BET	Betweenness centrality	Quantifies how often a function appears on shortest paths between other function pairs
DEG	Degree centrality	Counts the number of direct caller and callee connections
PAGE	PageRank	Assesses function influence based on the importance of its callers
random	Random	Randomly selects functions without considering graph structure

per strategy using five different approaches: four commonly used centrality metrics (closeness centrality, betweenness centrality, degree centrality, and PageRank) and random selection (summarized in Table 3). We then instructed the LLM to generate test cases to reach those functions. The results revealed that different centrality metrics excel under distinct structural conditions. Based on these findings, we developed automated decision rules for PILOT's target selection mechanism that select the most effective strategy among the five approaches for each program based on its structural characteristics. Section 5.4.2 presents the detailed analysis and validation of this mechanism's contribution to coverage improvement.

Evaluation Metrics. We evaluate PILOT and the baselines based on the following metrics:

- Vulnerability detection: Number of vulnerabilities that are detected by PILOT
- Edge coverage: Number of edges in the program's Control Flow Graph (CFG) executed by the fuzz inputs

5. Evaluation

We address the following research questions:

- **RQ1** [Zero-day Vulnerabilities] How many zero-day vulnerabilities can PILOT discover?
- **RQ2** [Vulnerability Detection] How does PILOT compare to baselines in terms of vulnerability detection?
- **RQ3** [Code Coverage] How does PILOT compare with baselines in terms of code coverage?
- **RQ4** [Ablation study] How does each strategy of PILOT contribute to the coverage metrics?
- **RQ5** [Cost] What are the costs of PILOT?

5.1. RQ1: Zero-day Vulnerabilities

To evaluate the effectiveness of discovering previously unknown vulnerabilities, we applied PILOT to all 43 programs at their latest versions.

Vulnerability Discovery. PILOT identified 51 previously unknown vulnerabilities across 24 programs. Table 4 summarizes the distribution of vulnerability types. The discovered vulnerabilities include 16 buffer overflows, 12 NULL pointer dereferences, 8 memory leaks, 4 out-of-bounds reads, 2 use-after-free cases, and several other logical flaws. Of these, 41 have been confirmed by maintainers, with 33 already fixed in subsequent releases. Three vulnerabilities have been assigned CVEs. The complete list of vulnerabilities with detailed information is provided in the artifact.

TABLE 4: [RQ1] Zero-day vulnerabilities discovered by PILOT.

Vulnerability type	Count	Confirmed	Fixed
Buffer overflow	16	14	11
NULL pointer dereference	12	10	8
Memory leak	8	4	3
Out-of-bounds read	4	4	2
Use-after-free	2	2	2
Invalid pointer access	2	2	2
Double-free	1	0	0
Division by zero	1	1	1
Integer overflow	1	1	1
Others[†]	4	3	3
Total	51	41	33

[†] Heap address leak (1), memory exhaustion (1), infinite loop (1), reachable assertion (1)

Representative Case Studies. To illustrate the types of vulnerabilities discovered by PILOT, we present two representative case studies: a buffer underflow in ImageMagick and a buffer overflow in YARA. We verified that state-of-the-art fuzzing approaches could not discover these vulnerabilities within the same time budget.

Case Study 1: Buffer Underflow in ImageMagick. PILOT identified a buffer underflow vulnerability (CVE-2025-XXXXX) in ImageMagick, specifically within the filename template processing logic of the `InterpretImageFilename()` function in `image.c`. The vulnerability stems from an incorrect assumption in the offset calculation mechanism. When format specifiers beginning with `%` are processed consecutively, the cumulative offset exceeds the template position difference, causing the buffer address calculation to underflow to a location preceding the buffer's start. Although not explicitly documented in the ImageMagick command documentation, such format specifiers are semantically necessary arguments for the filename templating functionality. PILOT included the `%` format specifier in its seed (e.g., `-format "%w x %h"`), enabling the discovery of this vulnerability that other approaches missed.

Case Study 2: Buffer Overflow in YARA. PILOT identified a heap buffer overflow vulnerability in YARA, specifically within the `yr_arena_load_stream()` function in `libyara/arena.c`. The vulnerability arises from integer underflow when validating buffer boundaries: small buffer sizes cause arithmetic underflow that bypasses size checks, enabling out-of-bounds memory access.

TABLE 5: [RQ2] Vulnerability detection performance comparison across 33 baseline programs (24-hour runs). Unique vulnerabilities refer to vulnerabilities not detected by any other method in this comparison.

	CF mutation			ZZ mutation		SF mutation	
	CarpetFuzz	**ProphetFuzz**	**PILOT-CF**	**ZigZagFuzz**	**PILOT-ZZ**	**SelectFuzz**	**PILOT-SF**
Total vulnerabilities	24	7	**33**	18	**25**	0	**3**
Unique vulnerabilities	18	4	27	4	6	0	1
Memory corruption	16	0	11	4	7	0	0
NULL pointer dereferences	10	6	18	11	10	0	1
Invalid operations	0	0	2	0	4	0	0
Resource management	0	1	1	2	3	0	2
Arithmetic/logic errors	0	0	1	1	1	0	0
Call chain depth	4.27 (±1.53)	4.11 (±1.21)	**4.79 (±2.91)**	2.22 (±1.03)	3.39 (±1.89)	0.00	2.00 (±1.41)

A critical challenge in reaching this function was that it only processes compiled YARA rule files (`.yarac`), which must be generated using the `yarac` compiler from source `.yar` files. Traditional seed generation approaches cannot orchestrate auxiliary tools like `yarac` to create the required input format. PILOT analyzed the call paths and identified this dependency, then generated a complete workflow: creating `.yar` rule files, compiling them with `yarac` to produce `.yarac` files, and invoking `yara -C` to load the compiled rules. PILOT selected this function as a target based on its high centrality in the call graph. Prior research [36] stated that high-centrality functions tend to contain more vulnerabilities, and this case validates that finding: the target function indeed contained a vulnerability.

Result 1. PILOT discovered 51 previously unknown vulnerabilities across 24 programs, with 41 confirmed and 33 fixed by developers, including three assigned CVE identifiers.

5.2. RQ2: Vulnerability Detection

To evaluate PILOT's vulnerability detection capability against prior work, we conducted a comparative evaluation using the same program versions from the POWER and CarpetFuzz datasets and configurations as baseline fuzzers.

Table 5 presents the vulnerability detection results across all baselines. PILOT consistently outperforms baselines in total vulnerabilities discovered: 33 vs 24 (CarpetFuzz), 33 vs 7 (ProphetFuzz), 25 vs 18 (ZigZagFuzz), and 3 vs 0 (SelectFuzz). PILOT discovers substantially more unique vulnerabilities, those not found by any other method in this comparison: 27 vs 18 (CarpetFuzz), 27 vs 4 (ProphetFuzz), 6 vs 4 (ZigZagFuzz), and 1 vs 0 (SelectFuzz). This demonstrates that PILOT explores distinct code regions that baseline approaches miss, rather than merely finding overlapping vulnerabilities.

PILOT's semantic exploration approach involves trade-offs with systematic numerical enumeration. For example, CarpetFuzz excels at discovering edge cases in specific numerical ranges through exhaustive exploration. It finds tiffcrop vulnerabilities by systematically testing coordinate combinations (e.g., `-z 1,1,2048,2048:1,2049,2048,4097`) and rotation angles (`-R 90`, `-R 270`). While PILOT fails to generate these precise numerical patterns, as illustrated in the case study in the previous section, it compensates by producing diverse semantically plausible inputs, enabling discovery of vulnerabilities in complex logic paths that numerical enumeration misses.

Beyond vulnerability quantity, PILOT reaches deeper code locations. Call chain depth in Table 5 measures the average number of function calls from the program entry point to the vulnerability location. PILOT-CF achieves an average depth of 4.79 (±2.91) compared to 4.27 (±1.53) for CarpetFuzz, with vulnerabilities discovered across a wider depth range (1-10 vs 1-7). The large standard deviation for PILOT-CF reflects its ability to discover vulnerabilities across a wider range of call depths (1 to 10) than CarpetFuzz (1 to 7). Similarly, PILOT-ZZ reaches depth 3.39 compared to 2.22 for ZigZagFuzz, consistently demonstrating PILOT's ability to explore deeply nested functions.

These results also reveal that the CarpetFuzz mutator reaches deeper code locations than the ZigZagFuzz mutator. This is because CarpetFuzz prepares multiple option seeds but does not mutate the strings themselves, thereby preserving the inter-relationships between options. Since PILOT generates option strings based on contextual understanding, its seeds are particularly effective when paired with mutators that maintain these option relationships, as evidenced by the superior performance of PILOT-CF over PILOT-ZZ.

Result 2. PILOT discovers 1.4-4.7× more total vulnerabilities and 1.5-6.8× more unique vulnerabilities than baselines, reaching deeper code locations through path-guided exploration of distinct code regions.

5.3. RQ3: Coverage Improvement

Fuzzing Coverage. Table 6 presents edge coverage results for a representative subset of 20 POWER baseline programs. Across the full 33-program benchmark suite, PILOT demonstrates superior coverage, achieving average improvements of 16.0% over CarpetFuzz, 36.2% over ProphetFuzz, 21.8%

TABLE 6: [RQ3] Edge coverage comparison across 20 POWER programs after 24 hours of fuzzing. Δ% indicates improvement of PILOT over the best-performing baseline (CarpetFuzz or ProphetFuzz for CF-based; ZigZagFuzz for ZZ-based; SelectFuzz for SS-based). "–" indicates that the baseline could not generate initial seeds due to insufficient information in man pages.

	CF mutation				ZZ mutation			SF mutation		
Program	**CarpetFuzz**	**ProphetFuzz**	**PILOT-CF**	**Δ%**	**ZigZagFuzz**	**PILOT-ZZ**	**Δ%**	**SelectFuzz**	**PILOT-SF**	**Δ%**
avconv	11,512	18,486	**34,788**	+88.2	10,633	**29,121**	+173.9	0	**395**	0
bison	5,869	4,029	**6,909**	+17.7	6,383	**7,264**	+13.8	2,464	**3,494**	+41.8
cflow	1,661	1,280	**1,692**	+1.9	2,088	**2,232**	+6.9	623	**726**	+16.5
cjpeg	1,095	1,133	**1,849**	+63.2	3,075	**6,072**	+97.5	34	**283**	+732.4
dwarfdump	6,470	5,007	**8,072**	+24.8	7,369	**10,389**	+41.0	53	**73**	+37.7
ffmpeg	22,637	22,839	**25,663**	+12.4	17,351	**37,729**	+117.4	10	**74**	+640.0
gm	6,216	5,465	**9,166**	+47.5	741	**10,580**	+1327.8	665	**3,607**	+442.4
gs	19,959	15,257	**20,304**	+1.7	15,149	**17,230**	+13.7	354	**1,428**	+303.4
jasper	2,169	—	**2,239**	+3.2	2,787	**3,153**	+13.1	12	**699**	+5725.0
mpg123	3,091	2,482	**4,111**	+33.0	3,627	**4,101**	+13.1	170	**416**	+144.7
nasm	7,103	4,850	**7,398**	+4.2	**8,183**	5,944	-27.4	781	**1,100**	+40.8
objdump	8,585	7,592	**12,957**	+50.9	6,274	**13,216**	+110.6	90	**2,180**	+2322.2
pspp	6,594	—	**10,486**	+59.0	2,606	**10,199**	+291.4	1,378	**3,272**	+137.4
readelf	8,291	5,906	**11,114**	+34.0	5,383	**11,597**	+115.4	1	**550**	+54900.0
tiff2pdf	3,350	3,788	**4,696**	+24.0	2,568	**5,708**	+122.3	1,029	**1,189**	+15.5
tiff2ps	2,550	2,697	**3,322**	+23.2	2,105	**3,221**	+53.0	**354**	191	-46.0
vim	27,886	11,961	**28,310**	+1.5	21,914	**36,159**	+65.0	**2,126**	1,875	-11.8
xmllint	9,233	6,269	**10,425**	+12.9	10,291	**16,652**	+61.8	656	**3,961**	+503.8
xmlwf	3,848	3,284	**4,233**	+10.0	2,855	**4,400**	+54.1	**173**	169	-2.3
yara	2,447	1,583	**2,817**	+15.1	2,441	**3,580**	+46.7	292	**702**	+140.4

over ZigZagFuzz, and 57.9% over SelectFuzz. The performance gap is particularly large for SelectFuzz. SelectFuzz uses distance metrics to guide mutations toward target locations, but struggles when initial seeds cannot reach the vicinity of target functions. CLI programs require specific option combinations to activate functionalities, creating all-or-nothing dependencies. Without initial seeds that satisfy these dependencies, SelectFuzz's distance-guided approach cannot compute meaningful guidance metrics, significantly limiting its effectiveness.

Initial Seed Coverage. To understand what drives these coverage improvements, we analyzed initial seed quality for each approach. Table 7 compares the characteristics of initial seed sets by executing each seed once.

The *minimal set* configuration represents the simplest possible invocation, containing only a single fixed option string, which is the typical approach used by many non-CLI-aware fuzzing frameworks [1]. For input files, we use the AFL++ fuzzer's default seed files for this minimal set configuration. Comparing against the minimal set (261 functions, 1,446 branches), we observe that CLI-aware approaches achieve substantially higher coverage. CarpetFuzz achieves 1.59× more functions, ProphetFuzz achieves 1.41×, and PILOT achieves 2.33×, demonstrating that systematic CLI option exploration is essential for reaching diverse program functionality.

Then, PILOT substantially outperforms all existing methods. Compared to CarpetFuzz, PILOT achieves 1.47× more function coverage, and 1.64× more branch coverage. Dictionary-based fuzzing performs worst with only 242 functions due to its limited predefined dictionary, while ProphetFuzz's LLM-guided approach achieves 369 functions but falls short of PILOT's coverage-guided approach.

To isolate individual contributions of CLI option generation and input file generation, we evaluate two ablated versions: fixed option (using PILOT's input files but single (minimal) option) achieves 340 functions, while fixed files (using PILOT's CLI generation but minimal seed files) achieves 509 functions. This reveals that CLI option generation contributes 1.96× improvement while input file generation contributes 1.79×. The combination produces synergistic benefits, with PILOT achieving the highest coverage and deepest call depth (4.77) across all metrics.

Table 8 provides a comprehensive view of seed generation across all 33 evaluated programs. PILOT discovers more CLI options than existing approaches (1,026 vs 629 for CarpetFuzz). Importantly, PILOT discovers 553 unique options not found by other approaches, demonstrating its superior ability to explore diverse CLI interfaces. Additionally, PILOT generates significantly more input files, producing an average of 204.3 files per program compared to 12.9 for CarpetFuzz and 44.1 for ProphetFuzz, which enables more comprehensive testing of file-dependent functionalities.

To illustrate PILOT's semantic advantage concretely, we categorize PILOT's unique options into three types:
(i) Standard functional options: PILOT extracts options directly from source code rather than relying on documentation, therefore it naturally discovers more options. For example, gs (Ghostscript) has extensive internal parameters for controlling rendering engine behavior, memory management, and color processing. PILOT discovers 214 options including these internal parameters such as `-dGraphicsAlphaBits` and `-dNumRenderingThreads`, while ProphetFuzz finds only 31 commonly documented options like `-sDEVICE` and `-sOutputFile`.

TABLE 7: Branch coverage, function coverage, and call depth statistics from execution of initial seeds.

Approach	Branches	Functions	Avg depth
Minimal set	1,446 ± 2,290	261 ± 435	4.21
Dictionary	1,365 ± 2,165	242 ± 415	3.82
CarpetFuzz	2,740 ± 3,499	414 ± 559	4.27
ProphetFuzz	2,506 ± 2,723	369 ± 420	4.68
PILOT	**4,487 ± 5,087**	**609 ± 794**	**4.77**
Fixed option	2,045 ± 2,076	340 ± 454	4.47
Fixed files	3,592 ± 5,352	509 ± 809	4.31

TABLE 8: Initial seed set characteristics: generated input files and discovered options.

Approach	Files	Options	Unique options
Dictionary	–	771	329
CarpetFuzz	12.9	629	176
ProphetFuzz	44.1	595	187
PILOT	**204.3**	**1,026**	**553**

(ii) Numeric boundary testing: PILOT generates rich numeric variations targeting edge cases. For example, in editcap, PILOT generates timestamps including `-2147483648` (Unix epoch minimum), `-999999999`, and `-86400`. This approach ensures comprehensive coverage of numeric edge cases.
(iii) Invalid option generation: PILOT deliberately generates invalid options to test error handling robustness (e.g., `--nonexistent` and `--unknown-param`). PILOT guides LLMs to analyze the control flow logic for reaching error handling functions.

Result 3. PILOT achieves substantial coverage improvements across the 33-program benchmark suite, with average gains of 16.0% over CarpetFuzz, 36.2% over ProphetFuzz, 21.8% over ZigZagFuzz, and 57.9% over SelectFuzz.

5.4. RQ4: Ablation Study

We conduct ablation studies to evaluate (1) the contribution of core PILOT components and (2) the effectiveness of different target selection strategies in improving coverage.

5.4.1. Core Component Contributions

We conduct ablation studies on 10 programs randomly selected from our benchmark suite, with 5 target functions for each program. We examined four configurations: without path-guided prompting, without iterative refinement, without native tool configuration, and with GPT-4.1 instead of Claude Sonnet 4.0. Table 9 shows the results.

W/o path-guided prompting. This configuration disables the path candidate summary prompting. On average, disabling the path-guided prompting resulted in a 19.8% reduction in function coverage and a 27.9% reduction in branch coverage across all programs. Without path guidance, PILOT must search the call sites of the target functions through the entire large codebase, significantly reducing its ability to generate effective test cases that cover the intended functionality. This is further evidenced by the function reachability dropping from 64% to 51%, indicating that path-guided prompting plays a crucial role in helping the LLM successfully reach the target functions.

W/o iterative refinement. This configuration disables PILOT's coverage-driven feedback loop, generating test cases only once without refinement based on coverage results. Instead of iteratively querying the LLM with feedback about which functions along candidate paths were covered, this variant generates all seeds in a single attempt. On average, removing iterative refinement resulted in a 17.9% reduction in function coverage and a 27.5% reduction in branch coverage across all programs.

W/o native tool configuration. This configuration removes the instruction to use native command-line tools for input file generation from the LLM prompts. This results in an average 11.4% reduction in function coverage and a 19.8% reduction in branch coverage, as the system cannot generate valid files for programs requiring specific file format inputs. Instead, it produces files with arbitrary bytes (e.g., `printf '\x00\x00\x00\x00' > input.bin`) that fail to satisfy format requirements.

Other LLM types beyond Claude. The default PILOT with Claude achieves an average function coverage of 506.5 functions, while PILOT-GPT achieves 361.3 functions. For branch coverage, PILOT achieves an average of 4,232 branches compared to PILOT-GPT's 1,907 branches. This consistent advantage stems from Claude Sonnet 4.0's strengths in two key areas: (1) advanced coding capabilities that enable more accurate test case generation, and (2) superior performance in agentic workflows that require iterative reasoning and tool use.

5.4.2. Effectiveness of Target Selection Strategies

To evaluate PILOT's call graph-based target selection strategy, we analyzed the relationship between graph structural properties and coverage performance.

Correlation analysis. To identify when centrality-based strategies outperform random selection, we conduct correlation analysis between call graph structure and strategy. For each of the 20 POWER programs, we measure branch coverage achieved by each strategy at a fixed token budget (one million tokens, averaged over three trials). We then compute each strategy's *advantage* over random selection: $A(p, s) = C(p, s) - C(p, \texttt{random})$, where $C(p, s)$ is the coverage achieved by strategy s on program p. A positive advantage indicates superior performance.

To understand what drives these advantages, we extract structural features from each program's call graph: basic metrics (nodes, edges, density), centrality distributions, and topological properties (clustering coefficient, diameter, strongly connected components). We then correlate these features with strategy advantages.

TABLE 9: [RQ4-1] Function and branch coverage for different configurations.

	W/o path-guided		W/o refinement		W/o config		PILOT-GPT		PILOT	
Program	**Function**	**Branch**	**Function**	**Branch**	**Function**	**Branch**	**Function**	**Branch**	**Function**	**Branch**
cjpeg	184	1,372	188	1,278	197	1,706	140	805	**205**	**2,068**
dwarfdump	628	4,164	615	3,806	808	5,178	543	3,058	**918**	**6,439**
gm	624	7,319	554	6,212	**743**	7,554	371	3,217	667	**8,290**
jasper	358	2,577	337	2,311	421	3,045	338	2,312	**428**	**3,257**
nasm	475	3,360	498	3,576	480	3,541	287	1,587	**565**	**4,348**
objdump	409	3,020	453	3,223	439	3,241	327	1,972	**607**	**5,068**
pspp	1,593	5,213	1,802	5,746	1,755	5,616	1,112	3,094	**1,904**	**6,311**
tiff2ps	184	1,291	193	1,230	201	1,535	189	1,339	**328**	**2,563**
xmlwf	160	1,190	140	1,080	162	1,267	124	747	**174**	**1,376**
cflow	184	965	197	1,080	256	1,557	182	935	**269**	**1,682**
Avg. change	-19.8%	-27.9%	-17.9%	-27.5%	-11.4%	-19.8%	-36.6%	-51.1%	–	–
Func. reachability	51%		54%		59%		49%		**64%**	

The correlation analysis reveals distinct structural preferences for each strategy (detailed in Appendix A). For example, CLOSE excels on programs with large diameter and long paths, where high-closeness functions bridge distant components. BET and DEG prefer concentrated PageRank distributions, succeeding when few functions dominate. PAGE favors fragmented graphs with small strongly connected components.

Strategy results. Based on the significant correlations, we generate rules for deciding strategies. For each strategy, we establish thresholds using median feature values where the strategy outperforms random selection, weighted by correlation strength (detailed in Appendix B). Our structural features successfully predict strategy effectiveness for all programs in the preliminary experiment. The average confidence score across recommendations was 0.80, indicating high prediction accuracy. PAGE is most frequently chosen (40 %), favoring fragmented call graphs where PageRank identifies important nodes in loosely connected structures. BET suits 35 % of programs with concentrated PageRank distributions, while CLOSE is optimal for 20 % with large diameters where distance-based centrality matters most.

Result 4. PILOT outperforms alternative configurations in the ablation study. These results demonstrate that PILOT's path-guided prompting, iterative feedback loop, native tool configuration, and adaptive target selection are essential components for achieving comprehensive program coverage.

5.5. RQ5: Cost Analysis

We analyzed the token usage and API costs across the POWER dataset (detailed results in Appendix C). The results reveal clear patterns in LLM resource consumption for automated test generation. On average, each program required 36 chat interactions with the LLM to complete the test generation process, consuming approximately 4.6 million input tokens and 90,000 output tokens per trial. The iterative nature of the exploration process, requiring 27-46 conversational exchanges per program, demonstrates how the LLM progressively builds understanding through multiple interactions with the codebase. The input-to-output token ratio averages approximately 2%, indicating that the LLM primarily operates in read mode to explore and gather necessary knowledge about the codebase, with relatively concise outputs for test case generation.

The average cost per program was $15.16 (USD), with individual costs ranging from $10.50 to $20.05 depending on program complexity and the extent of code exploration required. While this represents a substantial computational investment, it remains economically feasible for practical vulnerability discovery.

6. Related Work

Command-line Fuzzing. AFLargv [37] extends AFL to support command-line argument fuzzing by processing command-line options using fixed-length data chunks. However, it applies traditional file-based fuzzers in a straightforward manner without modifying the core AFL fuzzing algorithm. TOFU [38] expands the fuzzing space to include command-line arguments and uses distance metrics to prioritize inputs that are closer to specified target basic blocks in the program. POWER [34] systematically explores command-line option configurations by constructing diverse option configurations and selecting maximally "distant" configurations based on function relevance. However, these approaches lack robust validity checking for generated option combinations, resulting in many invalid configurations that lead to early program termination.

CrFuzz [5] introduced clustering analysis to predict input validity, demonstrating improvements in path and edge coverage. ConfigFuzz [6] implements a transformation technique that encodes program configurations as part of fuzzable input, allowing existing mutation operators to test program settings alongside normal inputs. CarpetFuzz [4] leverages natural language processing to automatically extract option constraints from documentation, significantly reducing the testing search space by filtering out invalid option combinations. ZigZagFuzz [1] improves fuzzing coverage by separately mutating command-line options and file

inputs in an interleaving manner. However, these studies do not prioritize option combinations based on their likelihood of improving coverage, leading to inefficient exploration of the vast option space.

ProphetFuzz [7] uses LLMs to automatically predict high-risk command-line option combinations from program documentation. ProphetFuzz prioritizes combinations more likely to contain vulnerabilities and generates semantically coherent commands with appropriate input files. However, ProphetFuzz has two fundamental limitations. First, ProphetFuzz relies solely on user manuals and online documentation to infer CLI options, causing problems when documentation is incomplete, outdated, or insufficient for understanding option interactions. Second, ProphetFuzz generates input files through generic Python scripts rather than dedicated commands, which leads LLMs to produce ad-hoc files with limited validity and variety. In contrast, PILOT analyzes source code to obtain accurate option information and provides a sandboxed shell environment where the LLM can use real file-generation tools, enabling the generation of valid and diverse command-line options that reflect each program's actual behavior and ultimately achieving substantially higher coverage.

Directed Fuzzing. Directed fuzzing such as Beacon [2] and SelectFuzz [3] guide fuzzing toward specific target locations. In particular, SelectFuzz employs a precondition-guided approach that selects seeds capable of satisfying branch conditions leading to target locations, improving the efficiency of reaching specific code regions. Other works such as Prospector [39], Liu et al. [36], Cerebro [40], She et al. [41], and Magneto [42] have explored prioritization strategies and call chain decomposition for directed fuzzing. However, these techniques target specific code locations through traditional input fuzzing and do not address the unique challenges of CLI programs where command-line option configurations significantly determine reachable program paths and vulnerability exposure.

LLM-based Test Generation. Numerous prior studies [43], [44], [45], [46], [47] have proposed methods for generating test cases using LLMs. TELPA [43] leverages program analysis to extract real usage scenarios and address hard-to-cover branches through feedback-based refinement. CoverUp [44] employs coverage metrics to iteratively guide LLMs toward improved line and branch coverage, significantly outperforming prior methods. MuTAP [45] utilizes mutation testing to identify weaknesses in generated tests and augment prompts with surviving mutants, enhancing bug detection capabilities. CITYWALK [48] specifically addresses C++'s complex features by incorporating project dependency analysis and language-specific knowledge to improve test correctness. These approaches demonstrate that combining structured program analysis with LLMs' generative capabilities yields the effective automated testing tools, but they focus on unit testing and operate at the function level, making them unsuited for exercising CLI behavior.

Fuzz4All [47] leverages LLMs as input generation and mutation engines to produce diverse, realistic inputs for multiple programming languages. Oliinyk et al. [49] presents research focused on embedded systems security, specifically targeting BusyBox, a ubiquitous software package in Linux-based embedded devices. However, these approaches rely on specific domain knowledge and cannot be universally applied to various programs.

7. Limitations

This section outlines the current limitations of PILOT and highlights the issues that require further investigation.

First, we have not investigated the optimal number of iterations for iterative refinement (Section 3.5). We currently set N_{trial} to 2, but this value may not be optimal for all programs and could depend on program complexity. However, increasing the number of iterations is constrained by substantial token usage, which accumulates over multiple iterations. This issue is exacerbated by PILOT's current approach of retaining the entire chat history for iterative refinement. Selective preservation of relevant context may be necessary to manage token costs efficiently and enable exploration of optimal iteration counts.

Second, we have not systematically evaluated PILOT across different LLM models. Our evaluation primarily uses Claude Sonnet 4.0 with limited comparison to GPT-4.1. We observed that even among commercial LLMs, model selection significantly impacts performance, suggesting that smaller or open-source models may yield substantially different results. Specifically, when using LLMs with limited reasoning capabilities and context windows, PILOT would likely experience degraded performance. Possible adaptations to support smaller models include decomposing prompts into smaller, focused subtasks that fit within limited context windows, using a pipeline architecture where separate LLM calls handle option extraction, file generation, and seed refinement independently, and employing retrieval-augmented generation (RAG) to provide relevant code snippets on-demand rather than loading entire codebases into context. However, whether these adaptations can match the performance of large commercial models remains an open question. Future work should systematically evaluate PILOT across different LLM scales and architectures to understand the performance-resource trade-offs and identify minimum capability requirements.

8. Conclusion

In this paper, we presented PILOT, a novel CLI fuzzing approach that leverages LLMs to generate semantically valid command-line option combinations and input files. PILOT combines path-guided context prompting, iterative refinement with coverage feedback, and autonomous orchestration of command-line tools for input file generation. Our evaluation on 43 real-world programs shows that PILOT discovered 51 previously unknown vulnerabilities, with 41 confirmed and 33 fixed by developers. These results demonstrate that our approach establishes a foundation for more effective and automated LLM-based CLI fuzzing methods.

References

[1] A. Lee, Y. Choi, S. Hong, Y. Kim, K. Cho, and M. Kim, "Zigzagfuzz: Interleaved fuzzing of program options and files," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–31, 2025.

[2] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 36–50.

[3] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2693–2707.

[4] D. Wang, Y. Li, Z. Zhang, and K. Chen, "{CarpetFuzz}: Automatic program option constraint extraction from documentation for fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1919–1936.

[5] S. Song, C. Song, Y. Jang, and B. Lee, "Crfuzz: Fuzzing multi-purpose programs through input validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 690–700.

[6] Z. Zhang, G. Klees, E. Wang, M. Hicks, and S. Wei, "Fuzzing configurations of program options," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–21, 2023.

[7] D. Wang, G. Zhou, L. Chen, D. Li, and Y. Miao, "Prophetfuzz: Fully automated prediction and fuzzing of high-risk option combinations with only documentation via large language model," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 735–749.

[8] FFmpeg Team, "FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video," https://ffmpeg.org, 2024, accessed: 2025-10-29.

[9] FFmpeg Project, "Ffmpeg protocols documentation," https://www.ffmpeg.org/ffmpeg-protocols.html, 2025, accessed: 2025-11-08.

[10] Anthropic, "Context window," https://support.anthropic.com/en/articles/7996848-how-large-is-claude-s-context-window, 2024.

[11] Libav Contributors, "libav: Multimedia processing library," git commit: c464278. [Online]. Available: https://github.com/libav/libav

[12] GNU Project, "Gnu bison: Parser generator." [Online]. Available: https://www.gnu.org/software/bison/

[13] ——, "Gnu cflow: C program flow analyzer." [Online]. Available: https://www.gnu.org/software/cflow/

[14] libjpeg-turbo Contributors, "libjpeg-turbo: Jpeg image codec." [Online]. Available: https://libjpeg-turbo.org/

[15] libdwarf Contributors, "libdwarf: Dwarf debugging information library." [Online]. Available: https://www.prevanders.net/dwarf.html

[16] FFmpeg Contributors, "Ffmpeg: Multimedia framework," version N-103440. [Online]. Available: https://ffmpeg.org/

[17] GraphicsMagick Contributors, "Graphicsmagick: Image processing system." [Online]. Available: http://www.graphicsmagick.org/

[18] Artifex Software, "Ghostscript: Postscript and pdf interpreter." [Online]. Available: https://www.ghostscript.com/

[19] mpg123 Contributors, "mpg123: Mpeg audio player and decoder." [Online]. Available: https://www.mpg123.de/

[20] NASM Contributors, "Nasm: Netwide assembler." [Online]. Available: https://www.nasm.us/

[21] GNU Project, "Gnu binutils: Binary utilities." [Online]. Available: https://www.gnu.org/software/binutils/

[22] ——, "Gnu pspp: Statistical analysis software." [Online]. Available: https://www.gnu.org/software/pspp/

[23] LibTIFF Contributors, "libtiff: Tiff library and utilities." [Online]. Available: https://gitlab.com/libtiff/libtiff

[24] B. Moolenaar and Vim Contributors, "Vim: Vi improved text editor." [Online]. Available: https://www.vim.org/

[25] GNOME Project, "libxml2: Xml c parser and toolkit." [Online]. Available: https://gitlab.gnome.org/GNOME/libxml2

[26] Expat Contributors, "Expat: Xml parser library." [Online]. Available: https://libexpat.github.io/

[27] VirusTotal, "Yara: Pattern matching tool." [Online]. Available: https://virustotal.github.io/yara/

[28] Anthropic, "Introducing claude 4," https://www.anthropic.com/news/claude-4, 2025.

[29] O. AI, "Gpt-4.1," https://openai.com/index/gpt-4-1/, 2024.

[30] clang 20.0.0git, "libclang: C interface to clang," https://clang.llvm.org/doxygen/group__CINDEX.html, 2025.

[31] "Networkx network analysis in python," https://networkx.org/documentation/stable/reference/index.html, 2025.

[32] Anthropic, "Strengthen guardrails," https://docs.anthropic.com/en/docs/test-and-evaluate/strengthen-guardrails/reduce-latency, 2025.

[33] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "{AddressSanitizer}: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[34] A. Lee, I. Ariq, Y. Kim, and M. Kim, "Power: Program option-aware fuzzer for high bug detection ability." in *ICST*, 2022, pp. 220–231.

[35] AFLplusplus Team, "AFLplusplus argv fuzzing," https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/argv_fuzzing, 2024, accessed: 2025-10-29.

[36] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1547–1559.

[37] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, Dec 2017, published 21 December 2017.

[38] Z. Wang, B. Liblit, and T. Reps, "Tofu: Target-oriented fuzzer," *arXiv preprint arXiv:2004.14375*, 2020.

[39] Z. Zhang, L. Chen, H. Wei, G. Shi, and D. Meng, "Prospector: Boosting directed greybox fuzzing for large-scale target sets with iterative prioritization," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1351–1363.

[40] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 533–544.

[41] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2194–2211.

[42] Z. Zhou, Y. Yang, S. Wu, Y. Huang, B. Chen, and X. Peng, "Magneto: A step-wise approach to exploit vulnerabilities in dependent libraries via llm-empowered directed fuzzing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1633–1644.

[43] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, "Enhancing llm-based test generation for hard-to-cover branches via program analysis," *arXiv preprint arXiv:2404.04966*, 2024.

[44] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," *arXiv preprint arXiv:2403.16218*, 2024.

[45] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, "Effective test generation using pre-trained large language models and mutation testing," *Information and Software Technology*, vol. 171, p. 107468, 2024.

[46] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 919–931.

[47] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[48] Y. Zhang, Q. Lu, K. Liu, W. Dou, J. Zhu, L. Qian, C. Zhang, Z. Lin, and J. Wei, "Citywalk: Enhancing llm-based c++ unit test generation via project-dependency awareness and language-specific knowledge," *arXiv preprint arXiv:2501.16155*, 2025.

[49] Y. Oliinyk, M. Scott, R. Tsang, C. Fang, H. Homayoun *et al.*, "Fuzzing {BusyBox}: Leveraging {LLM} and crash reuse for embedded bug unearthing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 883–900.

Appendix

1. Correlation Analysis Details

This appendix provides detailed correlation analysis between graph structural properties and coverage performance for each target selection strategy discussed in Section 5.4.2. Table 10 shows the complete correlation coefficients and p-values for all strategies across different graph metrics.

TABLE 10: [RQ4-2] Significant correlations between graph structural features and relative branch coverage advantages over random selection.

Strategy	**Structural feature**	**Pearson r**	**p-value**
CLOSE	diameter	0.525	0.018
CLOSE	avg_shortest_path	0.472	0.036
BET	pagerank_top10_concentration	0.462	0.040
BET	pagerank_gini	0.461	0.041
DEG	closeness_centrality_skew	0.457	0.043
CLOSE	closeness_centrality_skew	0.426	0.061
CLOSE	largest_scc_size	-0.420	0.065
DEG	pagerank_top10_concentration	0.399	0.081
CLOSE	largest_scc_ratio	-0.397	0.083
DEG	pagerank_gini	0.392	0.087
PAGE	largest_scc_size	-0.392	0.087
DEG	diameter	0.389	0.090
PAGE	largest_scc_ratio	-0.388	0.091

2. Decision Rule Generation

Based on the significant correlations, we automatically generate decision rules for recommending strategies. For each strategy s and significantly correlated feature f, we establish a threshold θ_f by computing the median value of f across programs where $A(p, s) > 0$.

For positive correlations, we recommend strategy s when $f(p) \geq \theta_f$; for negative correlations, when $f(p) \leq \theta_f$. Each decision rule is weighted by $|r|$ (absolute correlation). The overall confidence for recommending strategy s to program p is:

$$\text{Confidence}(p, s) = \frac{\sum_{i \in M_s} |r_i|}{\sum_{j \in R_s} |r_j|} \tag{1}$$

where M_s is the set of matched conditions and R_s is all conditions for strategy s. Table 11 summarizes the generated decision rules. Table 12 shows the distribution of chosen strategies for the 20 POWER dataset programs.

3. Detailed Cost Analysis

Table 13 presents the complete token usage and API costs for each program in the POWER dataset.

TABLE 11: Generated decision rules for strategy recommendation

Strategy	Feature	Threshold	$\lvert r\rvert$	Condition
CLOSE	diameter	$\geq$ 10.0	0.525	Large diameter
CLOSE	avg_shortest_path	$\geq$ 4.32	0.472	Long paths
CLOSE	closeness_skew	$\geq$ 5.22	0.426	Skewed closeness
CLOSE	largest_scc_size	$\leq$ 3.0	0.420	Small SCCs
CLOSE	largest_scc_ratio	$\leq$ 0.009	0.397	Fragmented
BET	pagerank_top10_conc.	$\geq$ 0.405	0.462	Concentrated PR
BET	pagerank_gini	$\geq$ 0.406	0.461	High PR inequality
BET	pagerank_skew	$\geq$ 8.18	0.376	Skewed PR
BET	density	$\leq$ 0.003	0.375	Sparse graph
BET	diameter	$\geq$ 10.0	0.353	Large diameter
DEG	closeness_skew	$\geq$ 5.22	0.457	Skewed closeness
DEG	pagerank_top10_conc.	$\geq$ 0.405	0.399	Concentrated PR
DEG	pagerank_gini	$\geq$ 0.406	0.392	High PR inequality
DEG	diameter	$\geq$ 10.0	0.389	Large diameter
DEG	pagerank_skew	$\geq$ 8.18	0.317	Skewed PR
PAGE	largest_scc_size	$\leq$ 3.0	0.392	Small SCCs
PAGE	largest_scc_ratio	$\leq$ 0.009	0.388	Fragmented

TABLE 12: Distribution of function selection strategies across POWER programs.

Strategy	**Count**	**%**	**Avg confidence**
PAGE	8	40.0%	0.88
BET	7	35.0%	0.80
CLOSE	4	20.0%	0.56
random	1	5.0%	0.77
Total	20	100%	0.80

TABLE 13: [RQ5] Input and output token usage and API costs for target programs.

Program	**# Chats**	**Input**	**Output**	**Cost (USD)**
avconv	38	4,897,240	81,451	$15.91
bison	40	5,064,940	121,440	$17.02
cflow	34	4,188,222	102,827	$14.11
cjpeg	33	4,260,867	72,569	$13.87
dwarfdump	46	6,195,032	97,352	$20.05
ffmpeg	40	5,193,340	100,943	$17.09
gm	40	5,307,676	92,965	$17.32
gs	42	5,497,537	73,344	$17.59
jasper	34	4,251,739	84,518	$14.02
mpg123	44	5,687,286	83,373	$18.31
nasm	29	3,200,466	60,541	$10.51
objdump	40	5,056,497	114,108	$16.88
pspp	27	3,252,209	69,922	$10.81
readelf	39	4,892,200	99,998	$16.18
tiff2pdf	24	2,898,758	120,321	$10.50
tiff2ps	27	3,277,745	113,568	$11.54
vim	40	4,640,882	77,075	$15.08
xmllint	43	5,349,191	82,767	$17.29
xmlwf	38	4,290,179	62,524	$13.81
yara	38	4,664,417	93,444	$15.39
Average	36	4,603,321	90,252	$15.16