Harnessing Large Language Models for Seed Generation in Greybox Fuzzing

Wenxuan Shi*
wenxuan.shi@northwestern.edu
Northwestern University

Yunhang Zhang*
u1399304@utah.edu
University of Utah

Xinyu Xing
xinyu.xing@northwestern.edu
Northwestern University

Jun Xu
junxzm@cs.utah.edu
University of Utah

ABSTRACT

Greybox fuzzing has emerged as a preferred technique for discovering software bugs, striking a balance between efficiency and depth of exploration. While research has focused on improving fuzzing techniques, the importance of high-quality initial seeds remains critical yet often overlooked. Existing methods for seed generation are limited, especially for programs with non-standard or custom input formats. Large Language Models (LLMs) has revolutionized numerous domains, showcasing unprecedented capabilities in understanding and generating complex patterns across various fields of knowledge. This paper introduces SeedMind, a novel system that leverages LLMs to boost greybox fuzzing through intelligent seed generation. Unlike previous approaches, SeedMind employs LLMs to create test case generators rather than directly producing test cases. Our approach implements an iterative, feedback-driven process that guides the LLM to progressively refine test case generation, aiming for increased code coverage depth and breadth. In developing SeedMind, we addressed key challenges including input format limitations, context window constraints, and ensuring consistent, progress-aware behavior. Intensive evaluations with real-world applications show that SeedMind effectively harnesses LLMs to generate high-quality test cases and facilitate fuzzing in bug finding, presenting utility comparable to human-created seeds and significantly outperforming the existing LLM-based solutions.

1 INTRODUCTION

To discover bugs in code, fuzzing [36] has been considered one of the most practical techniques, thanks to its easy application to production-grade software. After decades of development, greybox fuzzing [18, 29, 35] has emerged as the most preferable option. By leveraging lightweight instrumentation to obtain runtime feedback for driving the exploration, greybox fuzzing strikes a balance between the efficiency of blackbox fuzzing [17] and the depth of whitebox fuzzing [20]. The arising of mature greybox fuzzers like AFL [13], AFL++ [12], Honggfuzz [23] is furthering the popularity.

Technically, greybox fuzzing starts with a set of initial test cases, called *seeds*, and iteratively derives new test cases from the seeds to expand code coverage. Thus, to achieve better results, *it is critical to prepare a set of high-quality seeds to bootstrap the fuzzing process.* For example, a seed already reaching the buggy site can significantly reduce the difficulty and time cost for fuzzing to trigger the bug. Yet, obtaining good seeds has not been easy.

In practice, the most common strategy to prepare seeds is through manual inspection of the program logic and construction of desired test cases. This strategy might work for programs taking input of popular formats (e.g., XML, HTML, PDF, etc.), as their test cases are widespread and can be easily collected. However, for programs with non-standard input formats or even self-customized formats, this strategy becomes unaffordable. An alternative solution is run *generators* that can produce test cases automatically. This only works when an applicable generator already exists. Otherwise, new generators must be built from scratch, which, as we will elaborate in §2.2, is often infeasible or impractical.

The recent development of AI, especially Large Language Models (LLMs) such as the Generative Pre-trained Transformer (GPT) [14], offers a new opportunity for generic, effortless seed preparation. Many LLMs (GitHub Copilot [16], Amazon CodeWhisperer [3], OpenAI's GPT series [41], Anthropic's Claude series [1], etc.), after pre-training on massive code, comments, and documentation, are excelling at code-centric tasks (code understanding, code summarization, code completion, code translation, etc.) [6, 33, 39, 54]. A straightforward application of the technology would be to run an LLM to analyze the target program for fuzzing and generate the test cases it expects.

Indeed, recent research has explored LLMs for seed generation in fuzzing [2, 30, 32, 47, 52, 57, 60]. The proposed methods have attempted using various inputs (target program code, example test cases, functionality specifications, documentation, etc.) and prompts to request test cases from the LLMs. They have demonstrated effectiveness within different domains (parsers [2], compilers [57], the Linux kernel [60], and other generic software [30]). However, they may have significantly undermined the potential of LLMs for seed generation due to failure to address several fundamental challenges. **(C1)** The LLM in use, even in their most recent versions, may not support many input formats. For instance, GPT-4o [40], OpenAI's new flagship model, refuses to generate binary representations because it is constrained to text formats. **(C2)** LLMs are restricted by their *context window*—the amount of token the model can handle from both its input and response [46]. Arbitrarily dumping information (e.g., the entire code base of the target program) to the LLMs can overflow the context window and fail the generation. **(C3)** LLMs are known to present unpredictable behaviors, which can impede the generation of test cases. **(C4)** The LLMs can lack a basic understanding of the progress. Thus, it may overlook an incomplete task or endlessly repeat an accomplished task.

*These authors contributed equally to this work.

In this paper, we present a system, SeedMind, as an attempt toward generic, effective LLM-based seed generation. SeedMind incorporates four ideas to address challenges **C1 - C4**. ❶ Inspired by a recent OSS-Fuzz extension [44], SeedMind instructs the LLM to construct a generator that can produce test cases instead of asking it to directly output test cases. The generator can be represented as pure text, which, when executed, can generate seeds in any format (**overcoming C1**). ❷ SeedMind devises coverage-guided evolution by incorporating a feedback-based loop to guide the LLM to improve the generator toward broader and deeper code coverage gradually. This enables guided progress, avoiding blind explorations (**overcoming C4**). ❸ SeedMind only provides the LLM with the context necessary to improve the generator rather than dumping everything (e.g., all the code of the target program), avoiding overflowing the context window (**overcoming C2**). ❹ SeedMind introduces state-driven realignment. Once observing LLM behaviors deviating from the expected state, SeedMind attempts to re-align the LLM in the right direction via behavior-amending instructions (**overcoming C3**).

To understand the utility of SeedMind, we have performed an intensive set of evaluations. ① We apply SeedMind to generate test cases for all the functional C and C++ targets included in the OSS-Fuzz project [22] (166 programs and 674 harnesses). It shows that SeedMind can generate seeds with a quality close to the human-created ones. Further, SeedMind significantly outperforms the existing LLM-based solution. ② We use SeedMind to prepare seeds for running AFL [13], AFL++ [12], Honggfuzz [23] on MAGMA [24], a ground-truth fuzzing evaluation suite based on real programs with real bugs. The results demonstrate that SeedMind is compared to human-created seeds in facilitating fuzzing to find bugs. Likewise, SeedMind beats the existing LLM-based solution to a remarkable extent. ③ We diversify the LLM used by SeedMind to span GPT-4o [40], GPT-3.5-Turbo [43], and Claude-3.5-Sonnet [4]. SeedMind presents effectiveness with all three models, showing its generality. ④ We measure the cost of SeedMind for seed generation. SeedMind can generate high-quality seeds for a fuzzing harness with an average cost of less than $0.5, showing its affordability. We have also applied SeedMind in DARPA and ARPA-H's Artificial Intelligence Cyber Challenge (AIxCC) [8], a world-level, cutting-edge competition on developing AI-powered solutions for securing critical software infrastructures. SeedMind aided us in becoming one of the leading teams.

In summary, our main contributions are as follows.

- We introduce SeedMind, a system to offer generic, effective LLM-based seed generation.
- We incorporate a group of new ideas into SeedMind for addressing the major, prevalent challenges encountered by LLM-based seed generation.
- We implement SeedMind and intensively evaluate SeedMind on standard benchmarks. The results show that SeedMind offers generic, effective, and economical seed generation.

2 BACKGROUND

2.1 Greybox Fuzzing

Greybox fuzzing [18, 29, 35] emerged in the mid-2000s as a balance between blackbox [17] and whitebox fuzzing [20]. The concept was popularized by tools such as American Fuzzy Lop (AFL) [13]. The key idea of greybox fuzzing is to use lightweight instrumentation to obtain runtime feedback (e.g., code coverage) during fuzzing and, guided by the feedback, pick and mutate the existing test cases to derive new ones. To close the loop, new test cases offering new contributions (e.g., covering new code) are preserved for future mutations.

A greybox fuzzing campaign usually starts with an initial corpus of test cases called *seeds*. The quality of seeds significantly influences the fuzzing performance. For instance, given seeds covering a large code region, the chance of fuzzing to discover more bugs will be much higher, and the time needed will be much shorter. Hence, preparing a set of high-quality seeds has become a de facto standard before launching greybox fuzzing.

2.2 Seed Generation

To prepare high-quality seeds, a common practice is to manually understand the target program and craft desired test cases. However, this is not ideal for the highly automated process of fuzzing. An alternative idea is to run *generators* that can produce test cases automatically. Yet, it faces the challenge of constructing a generator when none is available.

To date, there are two main methods for constructing generators. ❶ Given the format specifications of input needed by the target program, people manually develop generators complying with the specifications to assemble test cases [15, 58, 61]. Those generators have mostly targeted standard formats like XML, HTML, and MathML, as their specifications are well documented. ❷ The other approach is based on machine learning techniques. After gathering a significant volume of inputs accepted by the target program, the method trains a model (probabilistic models [53], recurrent neural networks [21], generative adversarial networks [34], etc.) to learn the input structures, which is then run to generate new input variants.

Evidently, the two methods above are not preferable. They still *mandate manual efforts* to engineer the generator or collect the training data. More fundamentally, *they lack generality*. When the target program requires input following a non-standard or brand-new format, both methods become impractical. On the one side, format documentation is absent. Reversing the program to infer the format and building generators accordingly is unaffordable. On the other hand, satisfactory inputs in the wild become rare. There are limited sources to build an effective training dataset.

2.3 AI for Code

The recent development of large language models (LLMs) [5] has revolutionized the capability of AI in understanding and generating

Prompt: *Give me the actual binary representation for "ELF"*

GPT-4o:
"*I understand you're asking for an actual binary sequence (not just a text-based representation of it). However, since I can only communicate using text, I cannot provide literal binary byte streams like a file would contain, as my output is constrained to text formats.*"

Figure 1: Illustration of challenges incurred by input formats.

structured and unstructured text. Many models (e.g., GitHub Copilot [16], OpenAI Codex [42], Amazon CodeWhisperer [3]) are specifically designed and adapted for working with code. They present unprecedented performance in various code-centric tasks (code summarization, code completion, code translation, etc.) [6, 33, 39, 54]. Even generic models like OpenAI's GPT family and Anthropic's Claude family, after pre-training on massive code, comments, and documentation, are excelling at those tasks [33].

The strong code capability of LLMs has spurred their use in seed generation for fuzzing [2, 30, 32, 47, 52, 57, 60]. The existing methods in this line provide various types of input (target program code, example test cases, functionality specifications, documentation, etc.) together with customized prompts to LLMs, asking the LLMs to output test cases. They have demonstrated effectiveness on programs in various domains, such as parsers [2], compilers [57], the Linux kernel [60], and other generic software [30]. Compared to the aforementioned approaches for seed generation, LLM-based methods require less human effort while offer better generality. Yet, the existing methods have overlooked several challenges we will discuss shortly in §3.2. As a result, they have insufficiently utilized the potential of LLMs.

3 PROBLEM AND CHALLENGES

3.1 Problem Statement

In this paper, we focus on exploiting LLMs for seed generation in greybox fuzzing. To better define the problem, we clarify our assumptions below:

- **Input:** We assume access to the source code of the target program. We also assume the entry point of the fuzzing target is specified (e.g., the entry function needed by libFuzzer [50]). The two assumptions are minimal for functional fuzzing. Compared to the previous methods, we enforce no extra requirements (e.g., the availability of functionality specifications and documentation), thus representing a more generic scenario.
- **LLM:** We only require black-box access to the LLM. That is, the LLM can take our prompts as inputs and send us responses. The responses can be purely text-based, and support for other formats (e.g., image, audio, video) is not demanded.
- **Goal:** Previous research on LLM-based seed generation usually follows a vague objective like generating more diverse test cases. We consider a more specific goal. *We aim to generate test cases maximizing the code coverage in the target program.* The rationale is that modern greybox fuzzing tools prevalently take code coverage as their top priority. Hence, fulfilling our goal will better assist with the fuzzing process.

3.2 Technical Challenges

LLM-based seed generation appears to be intuitive. We may engineer a prompt, enclosing the target program code and the entry point information, to ask the LLM to produce diversified test cases. Indeed, previous research [32] has applied this idea. However, it tremendously undermines the potential of LLMs for seed generation, due to its failure to address several fundamental challenges.

Heterogeneous Input Formats (C1): The target programs of fuzzing can require a variety of input formats. For instance, the OSS-Fuzz project [49] has included 1260 open-source programs, spanning 130 input formats (text, image, video, audio, binary, etc.). *The LLM in use, even in their most recent versions, may not support many of those formats.* For instance, Figure 1 illustrates that GPT-4o [40], OpenAI's new flagship model, refuses to generate binary representations because it is constrained to text formats. In short, asking the LLM to directly generate desired test cases can fail.

Limited Context Window (C2): Restricted by computational resources and model architecture designs, LLM typically enforces a *context window*—the amount of token the model can handle from both its input and response [46]. For instance, GPT-4o has a context window of 128k tokens, while GPT-3.5-Turbo only has 16k. *Larger programs, such as server programs and OS kernels, can easily have a code size exceeding the context window, failing to be consumed by the LLMs.* Even given a sufficiently large context window, it is unwise to feed all code to the LLM, as a longer context can degrade the LLM's reasoning capability [56].

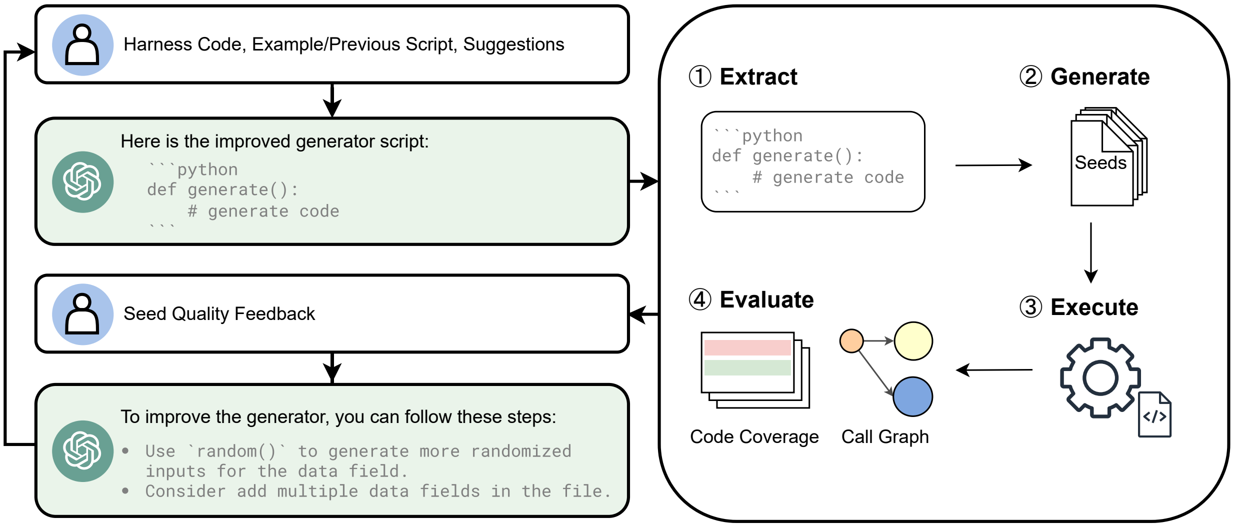
Unpredictable LLM Behaviors (C3): LLM is known to present unpredictable behaviors. *Such behaviors can impede the generation of test cases.* For instance, the LLM can run into hallucinations [62], generating test cases that seem plausible but unacceptable to the target program. It can also carry bias [27], preferring test cases with identical properties and compromising the diversity.

Blind Space Exploration (C4): In essence, the LLM needs to explore the code space of the target program and generate test cases covering different code blocks. Yet, by merely looking at the code, *the LLM lacks a basic understanding of the progress.* It may mistakenly believe a new block has been covered and skip it, or it may endlessly work on a block even if it has been covered, leading to limited yields or a waste of resources.

4 DESIGN AND IMPLEMENTATION

4.1 Overview

In this section, we present our system, SeedMind, as the first attempt to address challenges **C1 - C4**. The workflow of SeedMind is shown in Figure 2. Instead of asking the LLM to directly output test cases, SeedMind instructs the model to construct a generator that can produce test cases, following inspiration from a recent OSS-Fuzz extension [44]. The generator, consisting of only code, can be represented as pure text. Yet, via execution, it can generate test



Here is the improved generator script:

~~~
```python
def generate():
    # generate code
```
~~~

To improve the generator, you can follow these steps:

- Use `random()` to generate more randomized inputs for the data field.
- Consider add multiple data fields in the file.

Figure 2: Workflow of SeedMind.

cases of any format. This overcomes LLM's restriction on response format (**addressing C1**).

At its core, SeedMind incorporates a feedback-based loop to guide the LLM to gradually improve the generator toward producing test cases with broader and deeper code coverage. This design enables the LLM to explore the code space of the target program in a guided manner, mitigating blind explorations (**addressing C4**). To avoid overflowing the context window, we only provide the LLM with the context necessary to improve the generator rather than dumping everything (e.g., all the code of the target program) to it (**addressing C2**).

While guiding the LLM to improve the generator, SeedMind tracks its behaviors. Once observing behaviors deviating from the expected state, SeedMind attempts to re-align the LLM in the right direction via behavior-amending instructions (**addressing C3**). In the following, we elaborate on the technical details of SeedMind.

4.2 Initial Generator

Given a fuzzing target, SeedMind starts with producing a basic while functional generator. It first identifies the entry function of fuzzing and extracts its source code. For instance, given a fuzzing target prepared as a libFuzzer harness (one of the standard and most popular formats today) [50], SeedMind considers `LLVMFuzzerTestOneInput()` as the entry function. Using an elaborately crafted prompt which includes the entry function as a piece of context, SeedMind requests the LLM to output a test case generator in Python.

The prompts SeedMind adopted are presented in Appendix 8.1. While prompt engineering is not a focus of our research, those prompts are the results of numerous adaptations and optimizations in AIxCC [8], a world-leading competition on developing AI-powered solutions for securing critical software infrastructures. They have presented both generality and robustness in various real-world applications.

LLMs occasionally fail to produce a parsable and executable generator. We consider this an unpredictable behavior, and we will explain how we address it shortly in §4.4.

4.3 Coverage-Guided Evolution

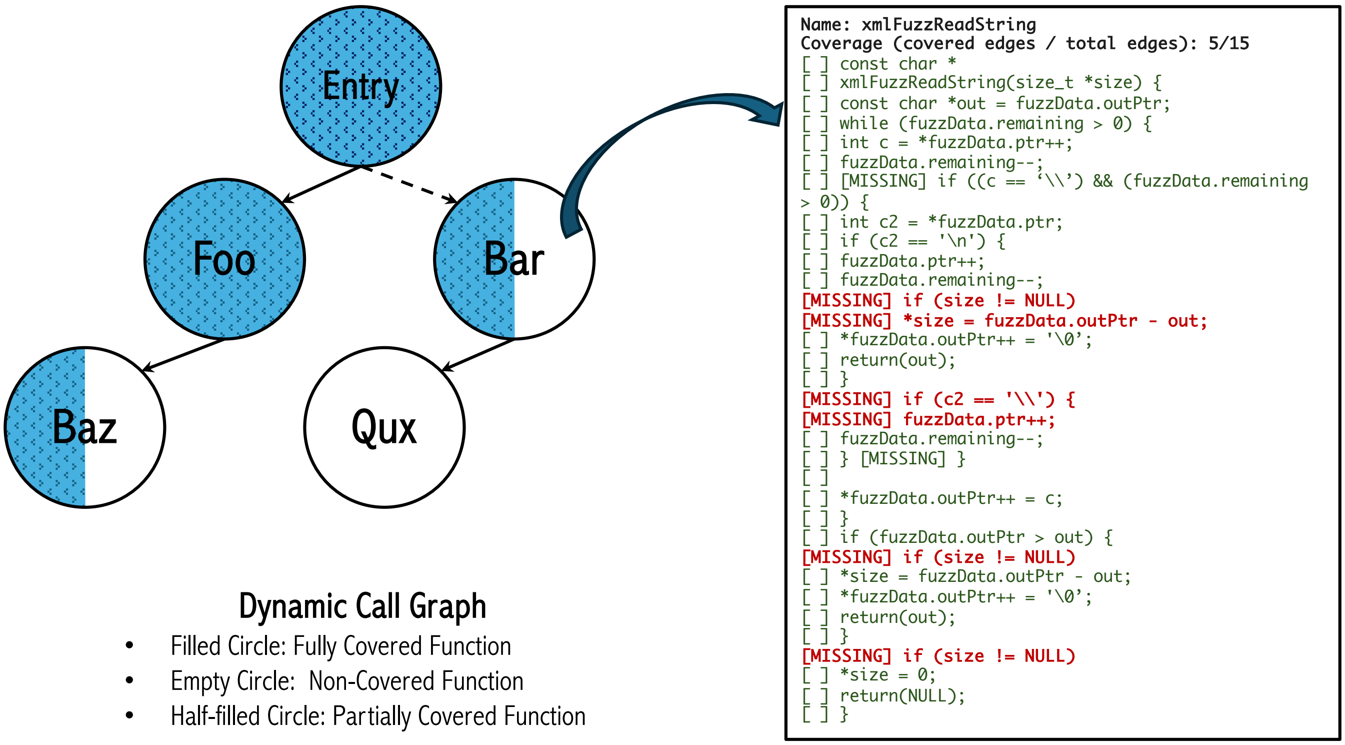
Even when the initial generator can run without issues, it oftentimes only generates test cases covering a limited region of code. We design a coverage-guided strategy to evolve the generator. Instead of aiming for a single, ultimate generator to reach maximal code coverage, we guide the LLM to iteratively create new variants to reach the non-covered code piece by piece.

Code Coverage Collection: Once a new working generator $\mathcal{G}$ is produced by the LLM, we run it to generate N (1,000 by default[1]) test cases. The code coverage of the test cases, quantified by branch coverage, is measured and merged with that of all previous generators. The total code coverage is represented on a *dynamic call graph* (i.e., the call graph composed of functions visited by all test cases from the existing generators and their children functions). Figure 3 illustrates this representation of code coverage.

Prompt Assemble: To evolve our generator $\mathcal{G}$, we aim to create a new variant that can reach the uncovered code segments. We use the most recent version of $\mathcal{G}$ along with the updated call graph and code coverage information to assemble a prompt. This prompt is then fed to the LLM as a feedback on its progress. However, a challenge arises: the call graph can be extensive, potentially resulting in a prompt that exceeds the LLM's context window limitations.

To address the challenge of context window limitations, we employ two key strategies for optimizing context usage. First, we focus exclusively on partially covered functions in the call graph (illustrated in the right side of Figure 3). We exclude fully covered and

[1] We observe that a single generator created by LLMs usually present limited diversity. Running it to generate 1,000 test cases can usually reach its maximal capability. 1,000 is also the default used by OSS-Fuzz's AI-based seed generator [44].


Entry
Foo
Bar
Baz
Qux
Dynamic Call Graph
• Filled Circle: Fully Covered Function
• Empty Circle: Non-Covered Function
• Half-filled Circle: Partially Covered Function

```
Name: xmlFuzzReadString
Coverage (covered edges / total edges): 5/15
[ ] const char *
[ ] xmlFuzzReadString(size_t *size) {
[ ] const char *out = fuzzData.outPtr;
[ ] while (fuzzData.remaining > 0) {
[ ] int c = *fuzzData.ptr++;
[ ] fuzzData.remaining--;
[ ] [MISSING] if ((c == ‘\\’) && (fuzzData.remaining
> 0)) {
[ ] int c2 = *fuzzData.ptr;
[ ] if (c2 == '\n') {
[ ] fuzzData.ptr++;
[ ] fuzzData.remaining--;
[MISSING] if (size != NULL)
[MISSING] *size = fuzzData.outPtr - out;
[ ] *fuzzData.outPtr++ = '\0’;
[ ] return(out);
[ ] }
[MISSING] if (c2 == '\\') {
[MISSING] fuzzData.ptr++;
[ ] fuzzData.remaining--;
[ ] } [MISSING] }
[ ]
[ ] *fuzzData.outPtr++ = c;
[ ] }
[ ] if (fuzzData.outPtr > out) {
[MISSING] if (size != NULL)
[ ] *size = fuzzData.outPtr - out;
[ ] *fuzzData.outPtr++ = '\0’;
[ ] return(out);
[ ] }
[MISSING] if (size != NULL)
[ ] *size = 0;
[ ] return(NULL);
[ ] }
```

Figure 3: An illustration of code coverage on dynamic call graph.

non-covered functions to reduce context size and maintain relevance. This exclusion is justified because (i) non-covered functions, being descendants of partially covered ones, do not contribute to covering missed branches in their predecessors, and (ii) $\mathcal{G}$ already incorporates knowledge about fully covered functions, which is provided to the LLM.

If the context still exceeds the LLM's capacity after this initial pruning, we implement an iterative pruning approach based on the dynamic call graph. Starting from the deepest level of the call graph, we progressively remove functions and move upwards until the token count falls within the specified limit. This method ensures efficient utilization of the available context while preserving the most critical functions in the program's execution flow.

New Generator Creation: After optimizing the context using the strategies described above, we feed the resulting prompt to the LLM to obtain suggestions for improving $\mathcal{G}$. This step is crucial as it guides the LLM to focus specifically on enhancing the generator's capabilities. An example of such a suggestion is illustrated in Figure 2. By explicitly requesting suggestions before asking for a new generator, we create a more focused chain-of-thought [55], ensuring the LLM concentrates on generator improvement rather than being distracted by other tasks.

These suggestions serve as valuable input for the subsequent iteration. We incorporate them into a new prompt, along with $\mathcal{G}$ and the partially covered functions identified earlier. This comprehensive prompt is then used to request an improved generator from the LLM.

4.4 State-Driven Realignment

LLMs can occasionally produce outputs that do not meet our requirements or exhibit unexpected behaviors. For instance, they may generate scripts that cannot be executed or fail to produce a script altogether. To address these issues, we employ a state-driven framework to regulate the system's behavior and ensure more consistent and reliable outputs.

As is shown in Figure 4, SeedMind employs a state machine to manage the progress of evolution. When the system detects unpredictable behaviors from the LLM, it initiates a realignment process with behavior-amending instructions. These instructions reverts the system to a previous state and prompting the LLM to rectify the error. For instance, if a generator script fails to execute, the system captures the standard error output and stack trace. This information is then fed back to the LLM with a request to diagnose and resolve the issue.

4.5 Implementation

SeedMind consists of two main components: an LLM agent and a runtime daemon. Our implementation comprises 5,162 lines of code in total, distributed across three programming languages: 2,096 lines of Go, 1,643 lines of Python, and 1,423 lines of Rust.

LLM Agent: The LLM agent is implemented in Python using the LangGraph framework. It operates as a state machine, with each state represented as a LangGraph node and transitions between states as edges in the graph. The state machine encompasses the

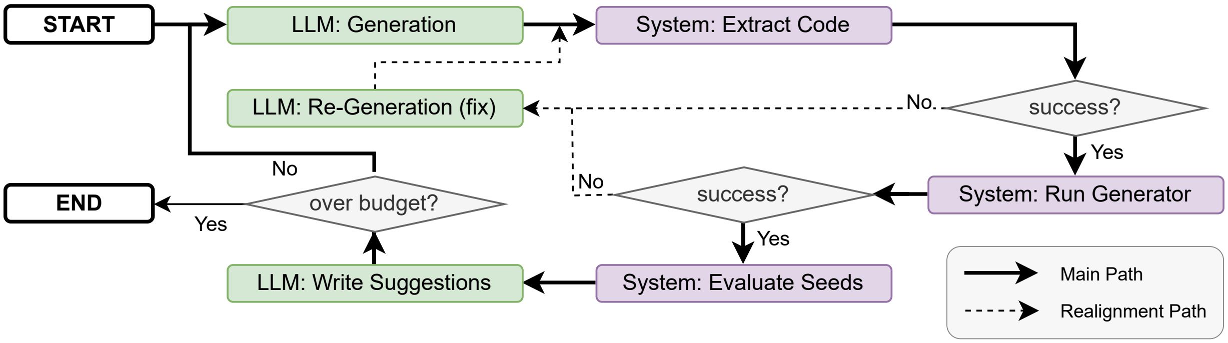


Figure 4: State Machine of SeedMind.

processes of initial generator creation, coverage-guided evolution, and state-driven realignment.

Runtime Daemon: The runtime daemon is written in Go and statically linked. It is designed for injection into isolated environments such as Docker containers used in OSS-Fuzz projects. The daemon's functions include compiling the target code, executing the target with generated seeds, collecting code coverage and function calling information, and generating the dynamic call graph.

Code Coverage and Call Graph Generation: Code coverage and function calling information are collected using multiple tools. A custom LLVM pass is used to instrument the program for dynamic function call data collection. Concurrently, LLVM's Coverage Sanitizer gathers code coverage data. To align this coverage information with the source code, tools such as `nm` and `llvm-symbolizer` are utilized to locate and extract relevant code snippets from source files.

Integration and Workflow: The LLM agent controls the overall process, determining generator improvements and realignment strategies. The runtime daemon executes these decisions in the target environment and provides coverage data and execution results back to the agent. Communication between the LLM agent and the runtime daemon is implemented using gRPC.

5 EVALUATION

To assess the utility of SeedMind, we perform a series of evaluations focusing on five questions:

① *Can* SeedMind *generate high-quality seeds?*

② *Can seeds generated by* SeedMind *facilitate fuzzing?*

③ *Can* SeedMind *outperform the existing LLM-based solutions?*

④ *What is the impact of the LLM used by* SeedMind*?*

⑤ *What is* SeedMind*'s cost to generate seeds for a fuzzing target?*

5.1 Experimental Setup

Benchmarks: To support our evaluation, we adopt two benchmarks. The first benchmark includes open-source programs collected from the OSS-Fuzz project [22]. We consider all the C and C++ programs in OSS-Fuzz. After excluding those without a seed corpus or unable to work with OSS-Fuzz's coverage utility[2], we end up with 166 programs. Each program configures one or more fuzzing harnesses following the format specified by libFuzzer [50]. In total, we include 674 harnesses. The goal of this benchmark is to asses whether SeedMind can generate high-quality test cases for a wide spectrum of programs.

The second benchmark is MAGMA [24], a ground-truth fuzzing evaluation suite based on real programs with real bugs. It includes 9 widely used open-source projects and 16 libFuzzer-style harnesses. We include this benchmark to measure how much SeedMind can facilitate fuzzing in discovering real-world bugs.

Baselines: We consider two baselines of seed preparation. The first baseline is the seed corpus shipped with both benchmarks[3]. The seeds were manually collected and maintained by the benchmark developers. They are also plentiful in number, ranging from dozens to thousands for each harness. This baseline can well represent the high-quality people manually prepare for greybox fuzzing. The second baseline is an AI-based seed generation solution Google recently extended for OSS-Fuzz [44]. It uses the basic idea of providing an LLM with the code of a fuzzing harness and asking the LLM to output test case generators in Python. This baseline represents a weaker version of LLM-based methods. For simplicity, we use OSSFuzz-AI to refer to this baseline. For the MAGMA benchmark, we additionally include an empty seed corpus as a third baseline, simulating a scenario where the fuzzing users do not prepare seeds.

Experiment Configurations: We conduct all experiments on CloudLab [11], with each machine equipped with Intel Skylake processors (20 physical cores @ 2.20GHz) and 192GB of RAM. For evaluations on the OSS-Fuzz benchmark, we run SeedMind and OSSFuzz-AI for 30 minutes on each fuzzing harness. We further limit the LLM API fees to $0.5 per hardness to avoid cost explosion[4]. On a specific harness, we run each generator to produce 1,000 test cases, and we set it to time out after 30 seconds to mitigate non-terminating generators.

[2] https://github.com/google/oss-fuzz/blob/master/infra/base-images/base-runner/coverage

[3] An example of default corpus shipped with OSS-Fuzz: https://github.com/DavidKorczynski/binary-samples;
An example of default corpus shipped with MAGMA: https://github.com/HexHive/magma/tree/v1.2/targets/libxml2/corpus/libxml2_xml_read_memory_fuzzer.

[4] We cannot restrict the API fees of OSSFuzz-AI as it uses Google's proxy, which offers no interfaces to retrieve the costs.

For evaluations on the MAGMA benchmark, we respectively run AFL [13], AFL++ [12], Honggfuzz [23] on each hardness with seeds from SeedMind and the three baselines (i.e., seeds generated by OSSFuzz-AI, the default seed corpus, and the empty seed corpus). Each harness is run for 24 hours with 3 instances affiliated to separate physical CPU cores. Each run is repeated 5 times to neutralize randomness.

To understand the impacts of different LLMs, we repeat all the OSS-Fuzz experiments with SeedMind and OSSFuzz-AI, separately using GPT-4o [40], GPT-3.5-Turbo [43], and Claude-3.5-Sonnet [4] as the LLM. We also employ a two-tier isolation strategy to ensure standardized and controlled testing conditions. First, for each fuzzing target, we run SeedMind in an isolated environment. This setup allows for independent execution of the LLM agent, enhancing security by isolating untrusted code generation. Second, each fuzzing target operates within its own Docker container, ensuring that each target has access to its required compilation environment and runtime libraries without interference from other targets.

Evaluation Metrics: For the OSS-Fuzz benchmark, we consider the code coverage, measured by branch coverage, of the seeds as the performance metric. For the MAGMA benchmark, we reuse the metrics recommended by the maintainers [24], including the time to reach a bug (TR) and the time to trigger a bug (TT).

5.2 Quality of Test Cases

To understand the quality of the test cases generated by SeedMind, we inspect their code coverage in the OSS-Fuzz programs. The detailed results are presented in Appendix §8.2. In summary, SeedMind can generate test cases to achieve satisfactory code coverage.

Comparing with Default Corpus: Among the 674 harnesses, SeedMind achieves greater code coverage than the default seed corpus on 48 harnesses when using GPT-3.5-Turbo, 253 harnesses with GPT-4o, and 268 harnesses with Claude-3.5-Sonnet. For these specific harnesses, SeedMind's code coverage is 43.0%, 44.1%, and 39.7% higher than the default seed corpus, respectively. Taking all the harnesses into account, SeedMind achieves 72.0%, 89.3%, 87.7% of the code coverage reached by the default seed corpus when using GPT-3.5-Turbo, GPT-4o, and Claude-3.5-Sonnet as the LLM. The results show that, *while not fully comparable, the seeds generated by* SeedMind *present quality close to that of the human-created corpus.* In many cases, SeedMind can even offer advantages.

Comparing with OSSFuzz-AI**:** Both as LLM-based solutions, SeedMind *significantly outperforms* OSSFuzz-AI. Regardless of which LLM is used, SeedMind achieves higher code coverage for a substantially larger number of harnesses than OSSFuzz-AI. With GPT-3.5-Turbo, SeedMind and OSSFuzz-AI can both run on 159 harnesses and SeedMind reaches higher code coverage on 142 of them. Switching to GPT-4o and Claude-3.5-Sonnet, SeedMind outperforms OSSFuzz-AI on 537 out of 636 harnesses and 588 out of 674 harnesses, respectively. If we only look at those harnesses, SeedMind presents code coverage 29.0%, 23.6%, 23.3% higher than OSSFuzz-AI. In the few instances where OSSFuzz-AI covers more code, the difference is mostly marginal and negligible. Thus, with all harnesses counted, SeedMind still shows code coverage 27.5%, 23.6%, 23.3% higher than OSSFuzz-AI across the three LLM configurations.

5.3 Benefits to Fuzzing

For assessing how much the seeds generated by SeedMind benefit fuzzing, we compare the results of SeedMind and the three baselines discussed in §5.1 on MAGMA. In this evaluation, we fix SeedMind and OSSFuzz-AI to use Claude-3.5-Sonnet because, as we will show shortly, Claude-3.5-Sonnet enables the best performance. Overall, SeedMind can facilitate the bug finding of all three fuzzing tools. For simplicity of presentation in the following, if a solution finds a bug using a time shorter than all other solutions, we call the bug a *fastest bug* found by that solution.

Comparing with Default Corpus: SeedMind appears comparable to the default seed corpus when applied for bug finding. Both present varying but close performances on different fuzzing tools. With AFL++, SeedMind enables the discovery of 27 bugs, and the default corpus enables 24. In addition, SeedMind finds 13 fastest bugs, while the default corpus finds 9. With Honggfuzz, the results are invested. The default corpus enables 24 bugs, with 14 fastest bugs. SeedMind only enables 21 bugs, with 7 fastest bugs. Their performance with AFL is more consistent. The default corpus enables more bugs (17 *v.s.* 14), but SeedMind finds more fastest bugs (9 *v.s.* 5).

Comparing with OSSFuzz-AI**:** SeedMind clearly beats OSSFuzz-AI. With AFL++, SeedMind enables the discovery of 9 more bugs (27 *v.s.* 18). SeedMind also finds bugs faster (13 fastest bugs *v.s.* 9 fastest bugs). The results with Honggfuzz are similar. The gap with AFL is smaller. They find the same amount of bugs, but SeedMind has a much faster pace (9 fastest bugs *v.s.* 4 fastest bugs).

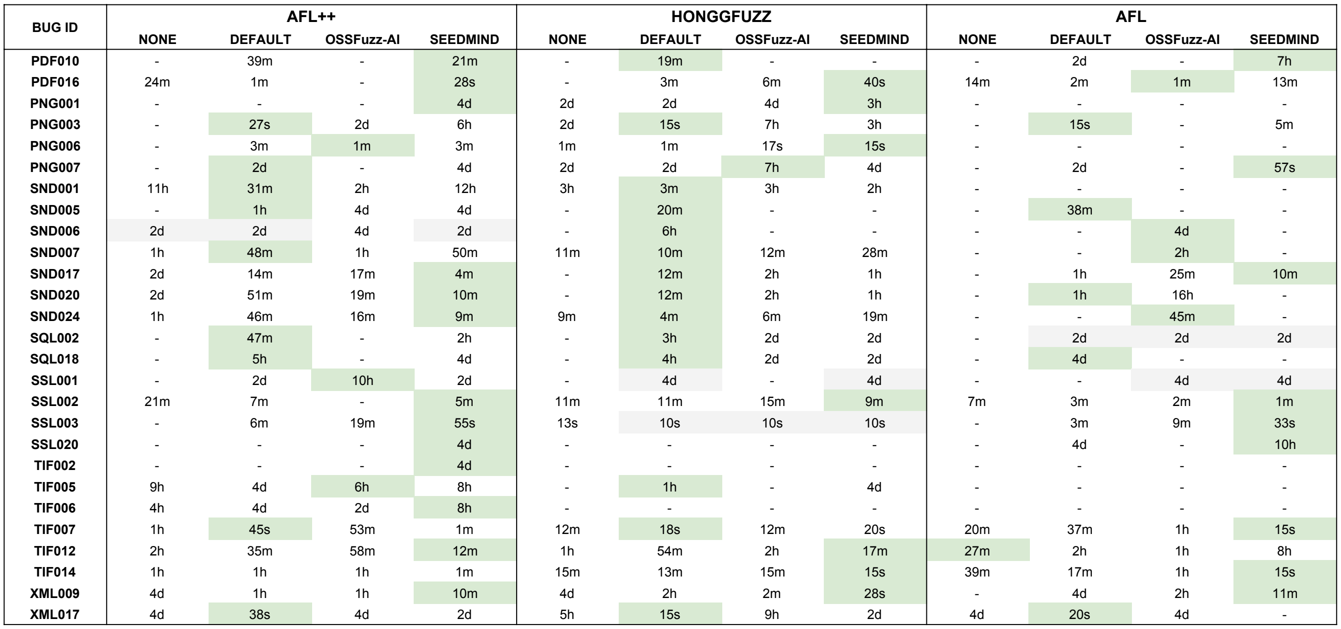
Comparing with Empty Corpus: SeedMind thoroughly defeats the empty corpus by consistently finding more bugs (27 *v.s.* 15 with AFL++, 21 *v.s.* 14 with Honggfuzz, and 14 *v.s.* 6 with AFL) at a fast pace (29 fastest bugs in total *v.s.* 1fastest bug in total). These show that SeedMind is a promising alternative when no default corpus is available.

5.4 Generality to LLM

To assess the generality of SeedMind across different language models, we conducted experiments using three tiers of LLMs: GPT-3.5-Turbo, GPT-4o, and Claude-3.5-Sonnet. Our results summarized in Table 1 demonstrate that SeedMind presents effectiveness with all three models, showcasing its adaptability to various LLM architectures and capabilities.

While SeedMind proves functional across all tested LLMs, we observed notable performance discrepancies. Claude-3.5-Sonnet results in superior performance, leading in code coverage for 385 harnesses and achieving an average code coverage of 18.03%. This is followed by GPT-4o, leading in 272 harnesses with an average coverage of 17.12%. In contrast, GPT-3.5-Turbo only achieves an average coverage of 15.12%, leading in 17 harnesses.

These results indicate that while SeedMind is effective with all tested LLMs, its performance can be enhanced by using more advanced models. We observed a positive correlation between the context window size and performance. Claude-3.5-Sonnet, with the



BUG ID	AFL++				HONGGFUZZ				AFL			
	NONE	DEFAULT	OSSFuzz-AI	SEEDMIND	NONE	DEFAULT	OSSFuzz-AI	SEEDMIND	NONE	DEFAULT	OSSFuzz-AI	SEEDMIND
PDF010	-	39m	-	21m	-	19m	-	-	-	2d	-	7h
PDF016	24m	1m	-	28s	-	3m	6m	40s	14m	2m	1m	13m
PNG001	-	-	-	4d	2d	2d	4d	3h	-	-	-	-
PNG003	-	27s	2d	6h	2d	15s	7h	3h	-	15s	-	5m
PNG006	-	3m	1m	3m	1m	1m	17s	15s	-	-	-	-
PNG007	-	2d	-	4d	2d	2d	7h	4d	-	2d	-	57s
SND001	11h	31m	2h	12h	3h	3m	3h	2h	-	-	-	-
SND005	-	1h	4d	4d	-	20m	-	-	-	38m	-	-
SND006	2d	2d	4d	2d	-	6h	-	-	-	-	4d	-
SND007	1h	48m	1h	50m	11m	10m	12m	28m	-	-	2h	-
SND017	2d	14m	17m	4m	-	12m	2h	1h	-	1h	25m	10m
SND020	2d	51m	19m	10m	-	12m	2h	1h	-	1h	16h	-
SND024	1h	46m	16m	9m	9m	4m	6m	19m	-	-	45m	-
SQL002	-	47m	-	2h	-	3h	2d	2d	-	2d	2d	2d
SQL018	-	5h	-	4d	-	4h	2d	2d	-	4d	-	-
SSL001	-	2d	10h	2d	-	4d	-	4d	-	-	4d	4d
SSL002	21m	7m	-	5m	11m	11m	15m	9m	7m	3m	2m	1m
SSL003	-	6m	19m	55s	13s	10s	10s	10s	-	3m	9m	33s
SSL020	-	-	-	4d	-	-	-	-	-	4d	-	10h
TIF002	-	-	-	4d	-	-	-	-	-	-	-	-
TIF005	9h	4d	6h	8h	-	1h	-	4d	-	-	-	-
TIF006	4h	4d	2d	8h	-	-	-	-	-	-	-	-
TIF007	1h	45s	53m	1m	12m	18s	12m	20s	20m	37m	1h	15s
TIF012	2h	35m	58m	12m	1h	54m	2h	17m	27m	2h	1h	8h
TIF014	1h	1h	1h	1m	15m	13m	15m	15s	39m	17m	1h	15s
XML009	4d	1h	1h	10m	4d	2h	2m	28s	-	4d	2h	11m
XML017	4d	38s	4d	2d	5h	15s	9h	2d	4d	20s	4d	-

Figure 5: Results of bug-finding evaluation with MAGMA. NONE means no seeds are used, and DEFAULT represents the default seed corpus shipped with the fuzzing target. The numbers stand for the average time-to-trigger of the corresponding bug. Values highlighted with green indicate the shortest time-to-trigger among the four solutions.

Table 1: Comparison of models

Model	Context Window	# of Highest Coverage	Average Coverage %
GPT-3.5-Turbo	16,385 tokens	17	15.12
GPT-4o	128,000 tokens	272	17.12
Claude-3.5-Sonnet	200,000 tokens	385	18.03

largest context window of 200,000 tokens, outperforms its counterparts, while GPT-3.5-Turbo, with the smallest window of 16,385 tokens, performs less. This is potentially because our pruning strategy described in §4.3 is applied more aggressively when the context window is smaller, as in GPT-3.5-Turbo.

Other factors may also contribute to these performance disparities. For example, the superior results of Claude-3.5-Sonnet and GPT-4o could be attributed to their lager model sizes and more diverse training datasets, enabling them to generate more effective and varied test cases.

5.5 Cost Analysis

We conducted a cost analysis to evaluate the economic feasibility of SeedMind for practical use. As explained before, we enforce a soft upper bound of $0.5 per harness to manage costs. This approach involves checking the accumulated cost after each iteration of seed generation. If the cost has not exceeded $0.5, the system continues to the next iteration. This method allows for slight budget overruns, ensuring that the last valuable iteration is not cut short.

Table 2 presents the average cost per fuzzing harness for each LLM, along with the number of harnesses that remained within our $0.5 bound. Claude-3.5-Sonnet strikes a balance between cost and performance, with an average cost of $0.48 per harness. It managed to stay within the $0.5 budget for 206 harnesses, the highest among all models, indicating its consistent performance across a wide range of scenarios. GPT-4o, while slightly exceeding our soft upper bound with an average cost of $0.69, remains acceptable given its strong performance.

It's noteworthy that for a significant number of harnesses, all models remained under the $0.5 threshold (49 for GPT-4o, 159 for GPT-3.5-Turbo, and 206 for Claude-3.5-Sonnet) within a strict 30-minute time limit. This suggests that SeedMind can be deployed cost-effectively for many fuzzing tasks, with the flexibility to allocate more resources to complex harnesses when necessary.

6 RELATED WORK

6.1 Seed Generation for Fuzzing

Generation-based fuzzing can produce highly structured inputs for real-world applications. Various approaches for structured test case generation have evolved over time:

Manually summarizing grammar rules. Generation-based fuzzers require well-written grammar rules prior to generating test cases. Examples of such fuzzers, designed for producing syntax-correct HTML files, include DOMATO [51], FREEDOM [58], and DOMFUZZ [38]. DOMFUZZ also employs a grammar-based splicing technique, which inspires our hierarchy object exchanging method.

Table 2: Average cost of running SeedMind to generate seeds for a fuzzing harness.

Model	Average Cost $	# of harnesses (<0.5$)
GPT-4o	0.69	49
GPT-3.5-Turbo	0.10	159
Claude-3.5-Sonnet	0.48	206

For fuzzing JavaScript codes, techniques like [45], [48], and [25] use random generation or combination of code based on provided syntax rules. Favocado [10] generates syntactically correct binding code for fuzzing JavaScript engines using semantic information.

Grammar generation with machine learning. Learn-&Fuzz [21] is a generation-based fuzzer that leverages machine learning to learn the grammar rules of PDF objects. However, it only generates random PDF objects and fails to capture the complexities of other elements in the PDF format, such as header, Xref, and trails. Skyfire [53] uses a context-sensitive grammar model with a probabilistic ML algorithm for fuzzing HTML and XSL files. DEPPFUZZ [31] employs a generative Sequence-to-Sequence model for C code generation, and Godefroid et al. [19] implement a dynamic test case generation algorithm for fuzzing IE7's JavaScript interpreter.

IR assisted generation. PolyGlot [7] is a fuzzing framework that creates high-quality test cases for different programming languages by using a uniform immediate representation (IR). Unlike other generation based fuzzing frameworks, PolyGlot uses grammar for mutation instead of pure seed generation, allowing for better code coverage. However, PolyGlot is limited by the requirement for a BNF grammar, and can still generate syntactically incorrect test cases due to inconsistent grammar inputs.

6.2 LLM-assisted Fuzzing

LLM-assisted seed generation. While traditional seed generation methods rely on manual grammar rules, machine learning, or intermediate representations, LLM-based approaches offer a more flexible and potentially more comprehensive solution for generating diverse, structure-aware seeds. CODAMOSA [28] leverages the code composition capabilities of LLMs to generate Python test cases specifically designed for fuzzing Python libraries and modules. Building on this concept, TITANFUZZ [9] extended the approach to generate API calls for deep learning software libraries. White fox [59] employs LLMs to analyze compiler-optimized code and generate test programs tailored for compiler optimization modules. These approaches demonstrate the potential of LLMs in seed generation, particularly for targets like interpreters and compilers that process program code as input. This alignment with LLMs' training data allows for high-quality seed generation and improved code coverage. However, these methods are often limited to specific domains or software types that primarily handle text-based inputs. In contrast, our system, SeedMind, offers a more versatile solution capable of generating seeds for a wide range of software types, including those that process non-textual inputs. This broader applicability makes SeedMind a more adaptable tool for fuzzing diverse real-world targets beyond just code-centric applications.

LLM-assisted seed mutation. CHATFUZZ [26] uses LLMs to mutate existing seeds in greybox fuzzing. It prompts ChatGPT to generate variations of seeds, aiming to produce format-conforming inputs that can pass initial parsing stages in programs expecting structured inputs. Similar to CHATFUZZ's approach, CHATAFL [37] extends the use of LLMs to protocol fuzzing. It enhances AFLNet by incorporating LLMs to extract machine-readable grammars for structure-aware mutation.

7 CONCLUSION

In this paper, we introduce SeedMind, a novel framework that utilizes Large Language Models for seed generation in greybox fuzzing. Unlike traditional approaches, SeedMind instructs LLMs to generate test case generators and iteratively refines them to expand code coverage. This approach systematically explores the target program, enhancing the fuzzing process. Our experiments show that SeedMind outperforms simpler AI-based methods and, in some cases, human-generated seeds. We assess SeedMind 's ability to produce high-quality seeds, its impact on fuzzing efficiency, and its generalizability beyond the LLM's training data. Overall, our results suggest that LLMs offer a promising solution for seed generation, with coverage feedback significantly improving seed quality and fuzzing effectiveness.

8 APPENDIX

8.1 Prompts Used by SeedMind

The system prompt, shown in Listing 1, defines the general role of the system. It sets the context for LLM, instructing it to act as a professional security engineer tasked with developing a Python script for generating test case files. This prompt is used at the beginning of each interaction to establish the LLM's role and primary objective.

```
SYSTEM_PROMPT = """
As a professional security engineer, your task is to develop a
      Python script that generates a new test case file. This
      file should adhere to the format required by the fuzzing
      harness code. The script will play a crucial role in
      creating diverse and effective test cases for thorough
      security testing.
"""
```

Listing 1: System prompt used by SeedMind.

The user prompt, shown in Listing 2, is used at the beginning of each iteration. It provides instructions for creating the Python script, including the harness code and basic script requirements. This prompt guides the LLM in generating a script that produces test cases compatible with the fuzzing harness while considering various input types, edge cases, and potential vulnerabilities.

```
USER_PROMPT = """
Write a Python script that generates a test case file
      compatible with the required format of the fuzzing
      harness code. The generated test cases should be diverse
      and effective for security testing purposes. Consider
      various input types, edge cases, and potential
      vulnerabilities relevant to the system being tested.
## Requirements for the Python Script:
```

```
- Generate data that the provided fuzzing harness code can use (focus on structure and file format).
- Avoid importing unofficial third-party Python modules.
## Fuzzing Harness Code:
{harness_code}
As an integrated component of an automated system, you should perform the tasks without seeking human confirmation or help.
## Instructions and Steps:
- You MUST ensure the python code is wrapped in triple backticks for proper formatting.
- You MUST include the full valid Python script in your response.
"""
```

Listing 2: User prompt used by SeedMind.

The example script prompt, shown in Listing 3, provides a template and specific requirements for the seed generation script. This prompt is used to guide the LLM in creating a script to generate a single test case and write it to an output file. It includes an example script to serve as a reference for the AI. Importantly, this prompt is used only once during the first iteration of the system. In subsequent iterations, it is replaced by the script generated from the previous iteration, allowing for continuous refinement and improvement of the seed generation process.

~~~
EXAMPLE_SCRIPT_PROMPT = """
The script should:
1. Has one argument, which is the output file path.
2. Generate one test case and write it to the output file.
3. The generated test case should be compatible with the fuzzing harness code provided.
Here is an example of Python script used to generate a testcase file:
```python
#!/usr/bin/env python3
import sys
import random
import base64
from typing import BinaryIO
def generate_input(rng: BinaryIO, out: BinaryIO, original_data: bytes):
    # original_data: constants data for your reference
    # random_num = rng.read(1)[0] % 100 + 1
    out.write(generated_data)
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python3 generate.py <output_file_path>")
        sys.exit(1)
    # replace it with constants that may be useful to the fuzzer
    original_data = b"0000"
    with open('/dev/urandom', 'rb') as rng, open(sys.argv[1], 'wb') as out:
        generate_input(rng, out, original_data)
"""
~~~

Listing 3: Example script prompt 1 used by SeedMind.

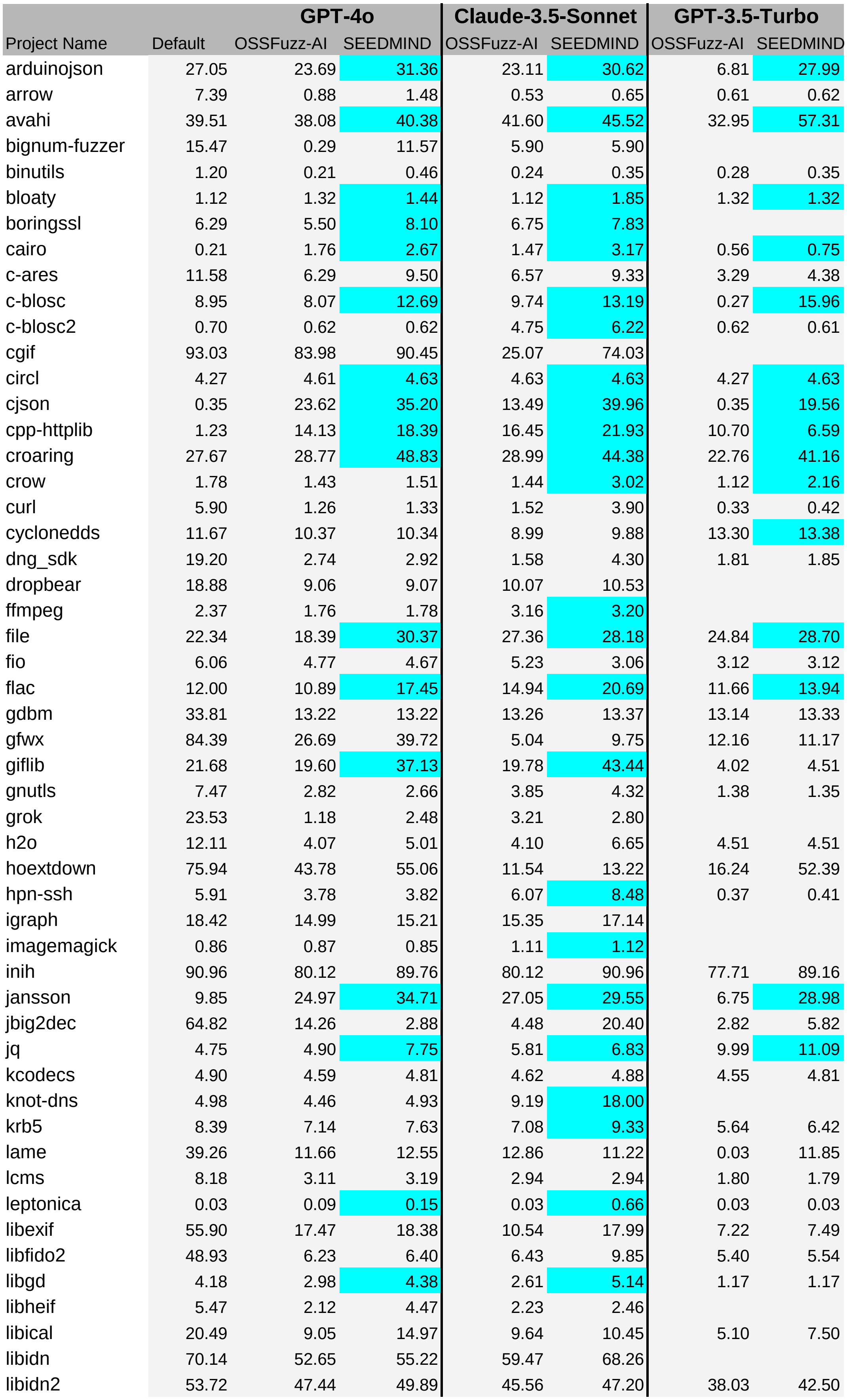
The summary prompt, presented in Listing 4, is used after generating and testing a script. It requests an analysis of the current generator based on coverage information. This prompt helps in evaluating the effectiveness of the generated script and provides guidance for improvements.

```
SUMMARY_PROMPT = """
Here is the coverage information for your generator. Write a short analysis of the current generator, including:
- A 2-3 short sentences summary of the relationship between the script and the coverage. For example, "The script not cover part X because it generates only Y type of data."
- A 2-3 short sentences general guideline on how to improve the script based on the coverage information received. You don't need to provide a new script, just some advice on how to improve the current one.
{coverage_report}
"""
```

Listing 4: Summary prompt used by SeedMind.

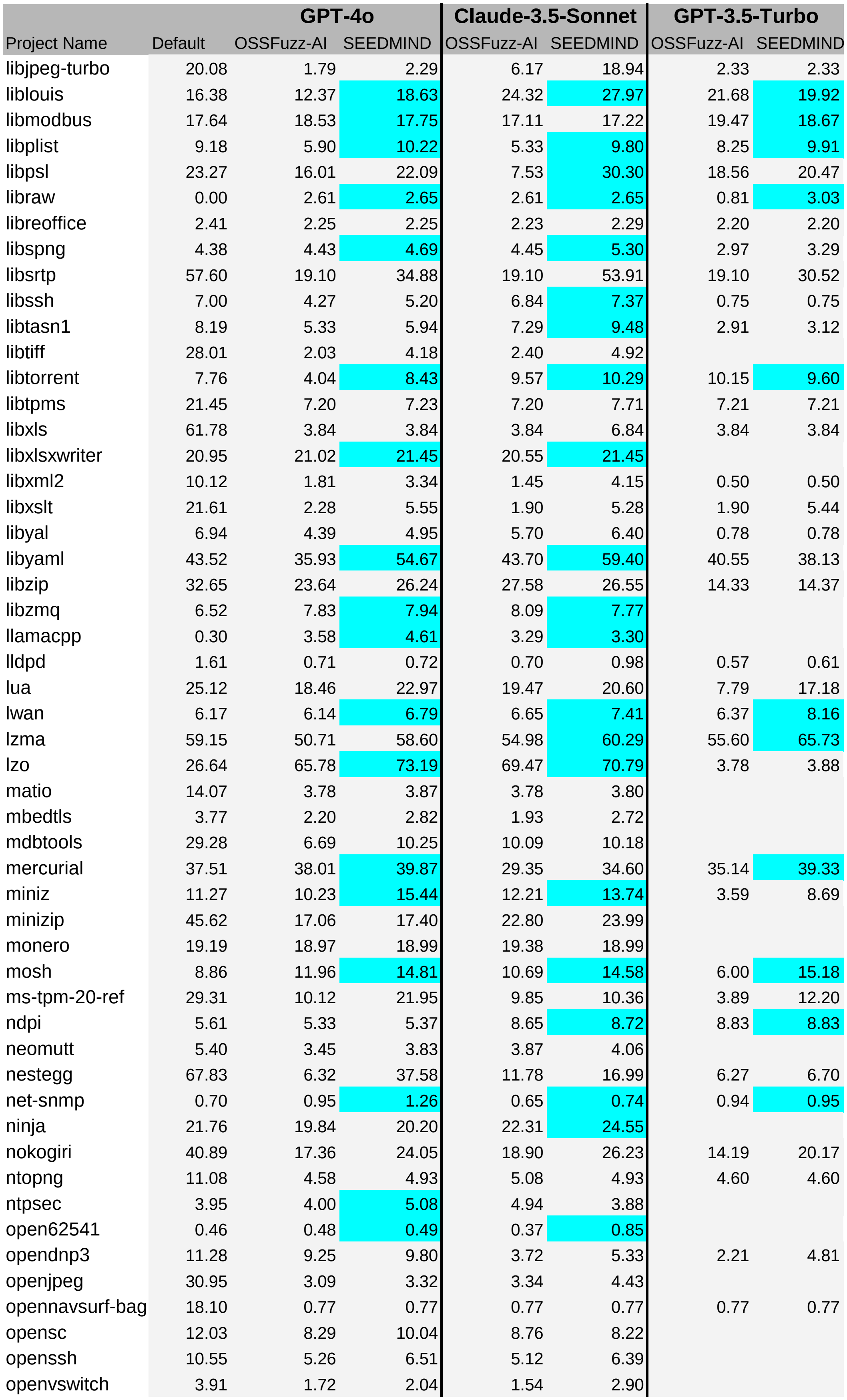
8.2 Code Coverage of OSS-Fuzz Programs

In Figure 6, Figure 7, and Figure 8, we present the code coverage results of test cases from different solutions on each OSS-Fuzz program.



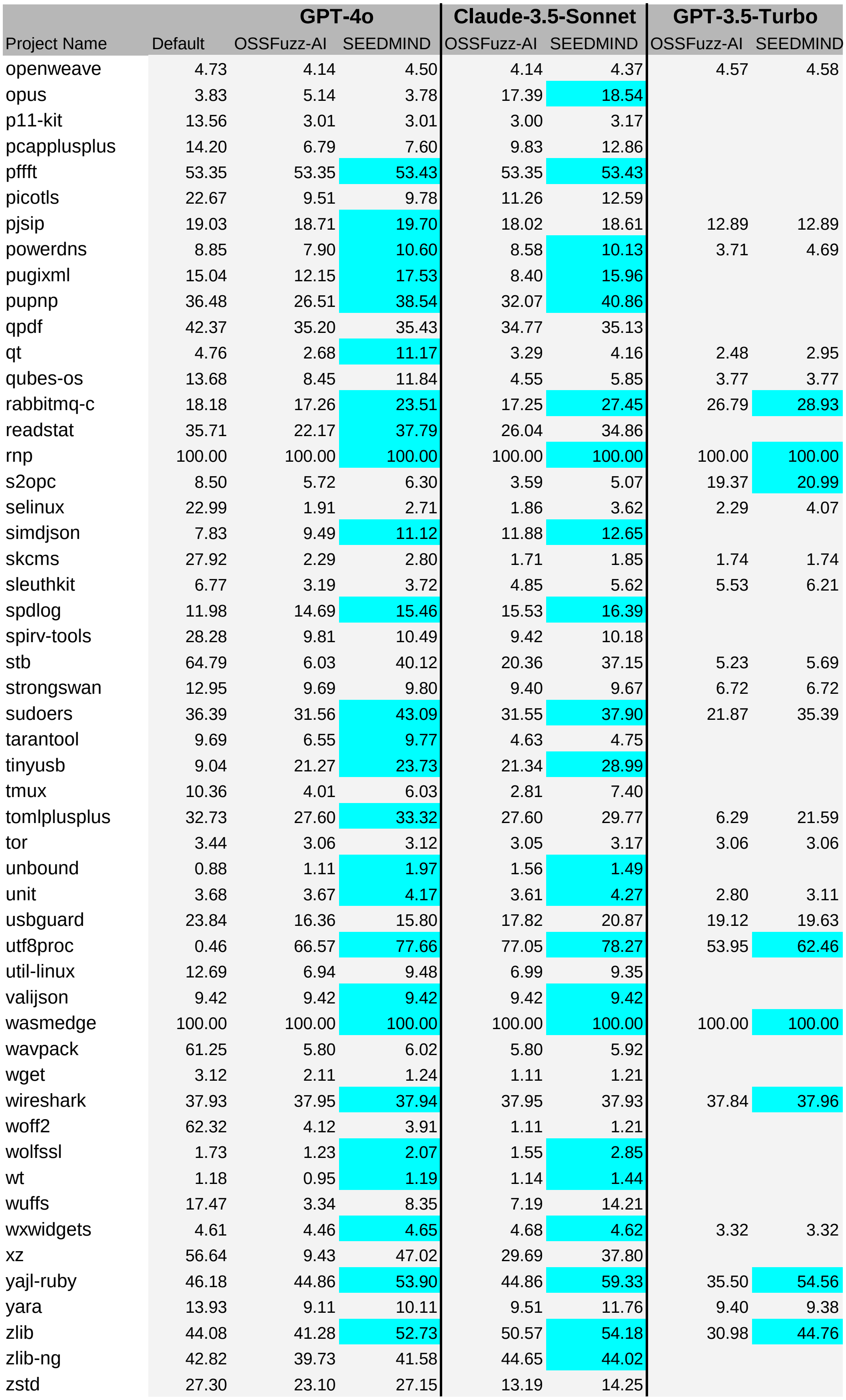
		GPT-4o		Claude-3.5-Sonnet		GPT-3.5-Turbo	
Project Name	Default	OSSFuzz-AI	SEEDMIND	OSSFuzz-AI	SEEDMIND	OSSFuzz-AI	SEEDMIND
arduinojson	27.05	23.69	**31.36**	23.11	**30.62**	6.81	**27.99**
arrow	7.39	0.88	1.48	0.53	0.65	0.61	0.62
avahi	39.51	38.08	**40.38**	41.60	**45.52**	32.95	**57.31**
bignum-fuzzer	15.47	0.29	11.57	5.90	5.90		
binutils	1.20	0.21	0.46	0.24	0.35	0.28	0.35
bloaty	1.12	1.32	**1.44**	1.12	**1.85**	1.32	**1.32**
boringssl	6.29	5.50	**8.10**	6.75	**7.83**		
cairo	0.21	1.76	**2.67**	1.47	**3.17**	0.56	**0.75**
c-ares	11.58	6.29	9.50	6.57	9.33	3.29	4.38
c-blosc	8.95	8.07	**12.69**	9.74	**13.19**	0.27	**15.96**
c-blosc2	0.70	0.62	0.62	4.75	**6.22**	0.62	0.61
cgif	93.03	83.98	90.45	25.07	74.03		
circl	4.27	4.61	**4.63**	4.63	**4.63**	4.27	**4.63**
cjson	0.35	23.62	**35.20**	13.49	**39.96**	0.35	**19.56**
cpp-httplib	1.23	14.13	**18.39**	16.45	**21.93**	10.70	**6.59**
croaring	27.67	28.77	**48.83**	28.99	**44.38**	22.76	**41.16**
crow	1.78	1.43	1.51	1.44	**3.02**	1.12	**2.16**
curl	5.90	1.26	1.33	1.52	3.90	0.33	0.42
cyclonedds	11.67	10.37	10.34	8.99	9.88	13.30	**13.38**
dng_sdk	19.20	2.74	2.92	1.58	4.30	1.81	1.85
dropbear	18.88	9.06	9.07	10.07	10.53		
ffmpeg	2.37	1.76	1.78	3.16	**3.20**		
file	22.34	18.39	**30.37**	27.36	**28.18**	24.84	**28.70**
fio	6.06	4.77	4.67	5.23	3.06	3.12	3.12
flac	12.00	10.89	**17.45**	14.94	**20.69**	11.66	**13.94**
gdbm	33.81	13.22	13.22	13.26	13.37	13.14	13.33
gfwx	84.39	26.69	39.72	5.04	9.75	12.16	11.17
giflib	21.68	19.60	**37.13**	19.78	**43.44**	4.02	4.51
gnutls	7.47	2.82	2.66	3.85	4.32	1.38	1.35
grok	23.53	1.18	2.48	3.21	2.80		
h2o	12.11	4.07	5.01	4.10	6.65	4.51	4.51
hoextdown	75.94	43.78	55.06	11.54	13.22	16.24	52.39
hpn-ssh	5.91	3.78	3.82	6.07	**8.48**	0.37	0.41
igraph	18.42	14.99	15.21	15.35	17.14		
imagemagick	0.86	0.87	0.85	1.11	**1.12**		
inih	90.96	80.12	89.76	80.12	90.96	77.71	89.16
jansson	9.85	24.97	**34.71**	27.05	**29.55**	6.75	**28.98**
jbig2dec	64.82	14.26	2.88	4.48	20.40	2.82	5.82
jq	4.75	4.90	**7.75**	5.81	**6.83**	9.99	**11.09**
kcodecs	4.90	4.59	4.81	4.62	4.88	4.55	4.81
knot-dns	4.98	4.46	4.93	9.19	**18.00**		
krb5	8.39	7.14	7.63	7.08	**9.33**	5.64	6.42
lame	39.26	11.66	12.55	12.86	11.22	0.03	11.85
lcms	8.18	3.11	3.19	2.94	2.94	1.80	1.79
leptonica	0.03	0.09	**0.15**	0.03	**0.66**	0.03	0.03
libexif	55.90	17.47	18.38	10.54	17.99	7.22	7.49
libfido2	48.93	6.23	6.40	6.43	9.85	5.40	5.54
libgd	4.18	2.98	**4.38**	2.61	**5.14**	1.17	1.17
libheif	5.47	2.12	4.47	2.23	2.46		
libical	20.49	9.05	14.97	9.64	10.45	5.10	7.50
libidn	70.14	52.65	55.22	59.47	68.26		
libidn2	53.72	47.44	49.89	45.56	47.20	38.03	42.50

Figure 6: Code coverage of test cases from different solutions on OSS-Fuzz programs. `Default` includes built-in seed corpus test cases. Values show the percentage of code base covered. For programs with multiple harnesses, results are averaged. Best results are in blue. If not highlighted, `Default` is the best.



		GPT-4o		Claude-3.5-Sonnet		GPT-3.5-Turbo	
Project Name	Default	OSSFuzz-AI	SEEDMIND	OSSFuzz-AI	SEEDMIND	OSSFuzz-AI	SEEDMIND
libjpeg-turbo	20.08	1.79	2.29	6.17	18.94	2.33	2.33
liblouis	16.38	12.37	18.63	24.32	27.97	21.68	19.92
libmodbus	17.64	18.53	17.75	17.11	17.22	19.47	18.67
libplist	9.18	5.90	10.22	5.33	9.80	8.25	9.91
libpsl	23.27	16.01	22.09	7.53	30.30	18.56	20.47
libraw	0.00	2.61	2.65	2.61	2.65	0.81	3.03
libreoffice	2.41	2.25	2.25	2.23	2.29	2.20	2.20
libspng	4.38	4.43	4.69	4.45	5.30	2.97	3.29
libsrtp	57.60	19.10	34.88	19.10	53.91	19.10	30.52
libssh	7.00	4.27	5.20	6.84	7.37	0.75	0.75
libtasn1	8.19	5.33	5.94	7.29	9.48	2.91	3.12
libtiff	28.01	2.03	4.18	2.40	4.92		
libtorrent	7.76	4.04	8.43	9.57	10.29	10.15	9.60
libtpms	21.45	7.20	7.23	7.20	7.71	7.21	7.21
libxls	61.78	3.84	3.84	3.84	6.84	3.84	3.84
libxlsxwriter	20.95	21.02	21.45	20.55	21.45		
libxml2	10.12	1.81	3.34	1.45	4.15	0.50	0.50
libxslt	21.61	2.28	5.55	1.90	5.28	1.90	5.44
libyal	6.94	4.39	4.95	5.70	6.40	0.78	0.78
libyaml	43.52	35.93	54.67	43.70	59.40	40.55	38.13
libzip	32.65	23.64	26.24	27.58	26.55	14.33	14.37
libzmq	6.52	7.83	7.94	8.09	7.77		
llamacpp	0.30	3.58	4.61	3.29	3.30		
lldpd	1.61	0.71	0.72	0.70	0.98	0.57	0.61
lua	25.12	18.46	22.97	19.47	20.60	7.79	17.18
lwan	6.17	6.14	6.79	6.65	7.41	6.37	8.16
lzma	59.15	50.71	58.60	54.98	60.29	55.60	65.73
lzo	26.64	65.78	73.19	69.47	70.79	3.78	3.88
matio	14.07	3.78	3.87	3.78	3.80		
mbedtls	3.77	2.20	2.82	1.93	2.72		
mdbtools	29.28	6.69	10.25	10.09	10.18		
mercurial	37.51	38.01	39.87	29.35	34.60	35.14	39.33
miniz	11.27	10.23	15.44	12.21	13.74	3.59	8.69
minizip	45.62	17.06	17.40	22.80	23.99		
monero	19.19	18.97	18.99	19.38	18.99		
mosh	8.86	11.96	14.81	10.69	14.58	6.00	15.18
ms-tpm-20-ref	29.31	10.12	21.95	9.85	10.36	3.89	12.20
ndpi	5.61	5.33	5.37	8.65	8.72	8.83	8.83
neomutt	5.40	3.45	3.83	3.87	4.06		
nestegg	67.83	6.32	37.58	11.78	16.99	6.27	6.70
net-snmp	0.70	0.95	1.26	0.65	0.74	0.94	0.95
ninja	21.76	19.84	20.20	22.31	24.55		
nokogiri	40.89	17.36	24.05	18.90	26.23	14.19	20.17
ntopng	11.08	4.58	4.93	5.08	4.93	4.60	4.60
ntpsec	3.95	4.00	5.08	4.94	3.88		
open62541	0.46	0.48	0.49	0.37	0.85		
opendnp3	11.28	9.25	9.80	3.72	5.33	2.21	4.81
openjpeg	30.95	3.09	3.32	3.34	4.43		
opennavsurf-bag	18.10	0.77	0.77	0.77	0.77	0.77	0.77
opensc	12.03	8.29	10.04	8.76	8.22		
openssh	10.55	5.26	6.51	5.12	6.39		
openvswitch	3.91	1.72	2.04	1.54	2.90		

Figure 7: Continued part of Figure 6.



		GPT-4o		Claude-3.5-Sonnet		GPT-3.5-Turbo	
Project Name	Default	OSSFuzz-AI	SEEDMIND	OSSFuzz-AI	SEEDMIND	OSSFuzz-AI	SEEDMIND
openweave	4.73	4.14	4.50	4.14	4.37	4.57	4.58
opus	3.83	5.14	3.78	17.39	18.54		
p11-kit	13.56	3.01	3.01	3.00	3.17		
pcapplusplus	14.20	6.79	7.60	9.83	12.86		
pffft	53.35	53.35	53.43	53.35	53.43		
picotls	22.67	9.51	9.78	11.26	12.59		
pjsip	19.03	18.71	19.70	18.02	18.61	12.89	12.89
powerdns	8.85	7.90	10.60	8.58	10.13	3.71	4.69
pugixml	15.04	12.15	17.53	8.40	15.96		
pupnp	36.48	26.51	38.54	32.07	40.86		
qpdf	42.37	35.20	35.43	34.77	35.13		
qt	4.76	2.68	11.17	3.29	4.16	2.48	2.95
qubes-os	13.68	8.45	11.84	4.55	5.85	3.77	3.77
rabbitmq-c	18.18	17.26	23.51	17.25	27.45	26.79	28.93
readstat	35.71	22.17	37.79	26.04	34.86		
rnp	100.00	100.00	100.00	100.00	100.00	100.00	100.00
s2opc	8.50	5.72	6.30	3.59	5.07	19.37	20.99
selinux	22.99	1.91	2.71	1.86	3.62	2.29	4.07
simdjson	7.83	9.49	11.12	11.88	12.65		
skcms	27.92	2.29	2.80	1.71	1.85	1.74	1.74
sleuthkit	6.77	3.19	3.72	4.85	5.62	5.53	6.21
spdlog	11.98	14.69	15.46	15.53	16.39		
spirv-tools	28.28	9.81	10.49	9.42	10.18		
stb	64.79	6.03	40.12	20.36	37.15	5.23	5.69
strongswan	12.95	9.69	9.80	9.40	9.67	6.72	6.72
sudoers	36.39	31.56	43.09	31.55	37.90	21.87	35.39
tarantool	9.69	6.55	9.77	4.63	4.75		
tinyusb	9.04	21.27	23.73	21.34	28.99		
tmux	10.36	4.01	6.03	2.81	7.40		
tomlplusplus	32.73	27.60	33.32	27.60	29.77	6.29	21.59
tor	3.44	3.06	3.12	3.05	3.17	3.06	3.06
unbound	0.88	1.11	1.97	1.56	1.49		
unit	3.68	3.67	4.17	3.61	4.27	2.80	3.11
usbguard	23.84	16.36	15.80	17.82	20.87	19.12	19.63
utf8proc	0.46	66.57	77.66	77.05	78.27	53.95	62.46
util-linux	12.69	6.94	9.48	6.99	9.35		
valijson	9.42	9.42	9.42	9.42	9.42		
wasmedge	100.00	100.00	100.00	100.00	100.00	100.00	100.00
wavpack	61.25	5.80	6.02	5.80	5.92		
wget	3.12	2.11	1.24	1.11	1.21		
wireshark	37.93	37.95	37.94	37.95	37.93	37.84	37.96
woff2	62.32	4.12	3.91	1.11	1.21		
wolfssl	1.73	1.23	2.07	1.55	2.85		
wt	1.18	0.95	1.19	1.14	1.44		
wuffs	17.47	3.34	8.35	7.19	14.21		
wxwidgets	4.61	4.46	4.65	4.68	4.62	3.32	3.32
xz	56.64	9.43	47.02	29.69	37.80		
yajl-ruby	46.18	44.86	53.90	44.86	59.33	35.50	54.56
yara	13.93	9.11	10.11	9.51	11.76	9.40	9.38
zlib	44.08	41.28	52.73	50.57	54.18	30.98	44.76
zlib-ng	42.82	39.73	41.58	44.65	44.02		
zstd	27.30	23.10	27.15	13.19	14.25		

Figure 8: Continued part of Figure 6.

REFERENCES

[1] 2024. Meet Claude \ Anthropic. https://www.anthropic.com/claude.

[2] Joshua Ackerman and George Cybenko. 2023. Large language models for fuzzing parsers (registered report). In *Proceedings of the 2nd International Fuzzing Workshop*. 31–38.

[3] Amazon. 2024. What is CodeWhisperer? https://docs.aws.amazon.com/codewhisperer/latest/userguide/what-is-cwspr.html.

[4] Anthropic. 2024. Introducing Claude 3.5 Sonnet \ Anthropic. https://www.anthropic.com/news/claude-3-5-sonnet.

[5] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2024. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* 15, 3 (2024), 1–45.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. *2021 IEEE Symposium on Security and Privacy (SP)* (2021), 642–658.

[8] DARPA. 2024. Artificial Intelligence Cyber Challenge. https://aicyberchallenge.com/.

[9] 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*.

[10] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *NDSS*.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*. 1–14.

[12] 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)*.

[13] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. 2023. Dissecting american fuzzy lop: a fuzzbench evaluation. *ACM transactions on software engineering and methodology* 32, 2 (2023), 1–26.

[14] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.

[15] Ivan Fratric. 2017. Domato. https://github.com/googleprojectzero/domato.

[16] GitHub. 2024. GitHub Copilot Your AI pair programmer. https://github.com/features/copilot.

[17] Patrice Godefroid. 2007. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. 1–1.

[18] Patrice Godefroid. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63, 2 (2020), 70–76.

[19] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 206–215.

[20] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.

[21] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.

[22] Google. 2024. OSS-Fuzz - continuous fuzzing for open source software. https://github.com/google/oss-fuzz.

[23] Google. 2024. Security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based). https://github.com/google/honggfuzz.

[24] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[25] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[26] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting Greybox Fuzzing with Generative AI. arXiv:2306.06782

[27] Dong Huang, Qingwen Bu, Jie Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2023. Bias assessment and mitigation in llm-based code generation. *arXiv preprint arXiv:2309.14345* (2023).

[28] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

[29] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1 (2018), 1–13.

[30] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. LLM-Powered Test Case Generation for Detecting Tricky Bugs. *arXiv preprint arXiv:2404.10304* (2024).

[31] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051.

[32] Darshan Lohiya, Monika Rani Golla, Sangharatna Godboley, and P Radha Krishna. 2024. Poster: gptCombFuzz: Combinatorial Oriented LLM Seed Generation for effective Fuzzing. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 438–441.

[33] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[34] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. 2018. Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606* (2018).

[35] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2018. The art, science, and engineering of fuzzing: A survey. *arXiv preprint arXiv:1812.00140* (2018).

[36] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[37] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model Guided Protocol Fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*.

[38] Mozilla Fuzzing Security. 2019. DOM fuzzers. https://github.com/MozillaSecurity/domfuzz.

[39] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[40] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/.

[41] OpenAI. 2024. Models. https://platform.openai.com/docs/models.

[42] OpenAI. 2024. OpenAI Codex. https://openai.com/index/openai-codex/.

[43] OpenAI. 2024. OpenAI's GPT-3.5 Turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.

[44] OSS-FUzz. 2024. Fuzz target generation using LLMs. https://github.com/google/oss-fuzz-gen/blob/main/llm_toolkit/corpus_generator.py.

[45] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1629–1642.

[46] Ryan Richards and Cal Wilmott. 2024. Understanding Large Language Models Context Windows | Appen. https://www.appen.com/blog/understanding-large-language-models-context-windows.

[47] Matthew J Rutherford and Alexander L Wolf. 2003. A case for test-code generation in model-driven systems. In *International Conference on Generative Programming and Component Engineering*. Springer, 377–396.

[48] Mozilla Fuzzing Security. 2019. A collection of fuzzers in a harness for testing the SpiderMonkey JavaScript engine. https://github.com/MozillaSecurity/funfuzz.

[49] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[50] Kostya Serebryany. 2024. A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[51] Symeon. 2020. Grammar based fuzzing PDFs with Domato. https://rb.gy/gi4cbz.

[52] Elwin Tamminga, Bouwko van der Meijs, and Ultraware Stjepan Picek. 2023. Utilizing Large Language Models for Fuzzing: A Novel Deep Learning Approach to Seed Generation. (2023).

[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[54] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[56] Natanael WF. 2024. Reasoning Degradation in LLMs with Long Context Windows: New Benchmarks. https://community.openai.com/t/reasoning-degradation-in-llms-with-long-context-windows-new-benchmarks/906891?page=2.

[57] 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.

[58] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 971–986.

[59] 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. In *OOPSLA 2024*.

[60] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563* (2023).

[61] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[62] Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, and Li Yuan. 2023. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469* (2023).