Branch Flipper: Unlocking Fuzz Blockers with Coverage-Grounded LLMs

Youngjoong Kim
Theori Inc.

Team U.S.
Theori Inc.

Team AI for Offensive Security
Theori Inc.

Abstract

Among various fuzzing techniques, coverage-guided fuzzing and greybox fuzzing aim to expand the test coverage by using it as a signal to select seed inputs for effective mutation and exploration. However, despite this feedback mechanism, the number of explorable code paths diminishes exponentially with increasing branching depth, making it challenging to cover deeper execution paths solely through random mutations. Besides, with the significant advancements in large language models (LLMs), fuzzing strategies that leverage LLMs have been explored recently, for seed and fuzz harness generation. In this paper, we propose a system *Branch Flipper* that localizes branch blockers and generates inputs to unlock them using LLMs. The proposed system consists of three modules: LLM-based seed generation, fuzz blocker localizer, and seed evaluation feedback loop. To support LLM-based binary input generation, we introduce Kaitai Struct, which enables binary generation and visualization to be more meaningful, allowing the LLM to more effectively generate binary inputs. For fuzz blocker localization, the system manages a set of blocker candidates and prioritizes them to resolve using the LLM sequentially. To mitigate LLM hallucinations and improve reliability, the system incorporates a grounding process using oracles and coverage-based feedback. Rather than solely relying on randomness, the system addresses branch blockers semantically, utilizing the reasoning capabilities of LLMs. We conduct experiments to test the validity of the system, and they demonstrate that the branch flipper effectively addresses fuzz blockers. Moreover, its capability of generating inputs that reach suspicious or potentially vulnerable functions also indicates its practical applicability to penetration testing contexts.

1 Introduction

Fuzzing is a software testing methodology that assesses software robustness by generating random program inputs to identify potential ill-behaviors. Among various approaches, greybox fuzzing utilizes code coverage to expand the testing scope by applying random mutations to seeds that previously increased coverage. However, random mutation alone poses limitations even under the coverage feedback. Particularly, randomly mutated inputs struggle to take the unhit branches of nested or complex predicates. In practice, greybox fuzzers used in projects like OSS-Fuzz [23] often exhibit coverage saturation, where further code paths are no longer being discovered despite continued testing. To address this, grammar-based fuzzing and similar techniques have been studied, but these typically require hand-crafted features and domain knowledge, making it difficult to scale fuzzing to a wide range of software targets.

Large Language Models, also referred to as LLMs [2], have shown a dramatic improvement in their natural language generation capabilities. With an instruction tuning [20], it has become possible to produce user-intended outputs even with zero or few samples [28]. By leveraging these capabilities, there has been a growing interest in applying LLMs to fuzzing. Recent studies have explored using LLMs to generate fuzz harnesses(drivers) [16, 23, 29], and to generate input seeds [7, 12, 18, 24, 25]. In the case of harness generation, however, evaluating whether the generated harnesses truly reflect the actual use cases remains a significant challenge. Even if a crash is triggered by the generated harness, it is often difficult to convincingly argue that the resulting crash is meaningful or to maintain consensus on its validity.

On the other hand, leveraging LLMs for seed generation also suffers from challenges. While random mutation can be performed and evaluated at a very high frequency, LLMs require high-performance hardware and thus suffer from the inevitable latency, making it impractical to completely replace random mutation with them. Even when using LLM APIs, the process can be costly in terms of both time and resources. As a result, there is a growing need for strategies that selectively incorporate LLMs.

Recently, there have been efforts to fully automate penetration testing using LLM-based agentic systems, such as ReAct [32]. These systems are capable of planning actions, sending queries, interpreting results, and managing history throughout the process [8, 13]. Furthermore, U.S. DARPA, Defense Advanced Research Projects Agency, launched the AI Cyber Challenge (AIxCC), a competition aimed at developing AI agents that automatically discover vulnerabilities in open-source software projects, which are widely and actively used, but their security and reliability are often treated as secondary concerns. Given the diverse scope of open-source projects and the broad range of potential vulnerabilities, the competition requires a general-purpose testing system that can effectively handle such diversity.

To support greybox fuzzing in various situations and complement its random mutation strategy, we propose a system, *Branch Flipper*, which aims to generate *blocker-unlocking* inputs when fuzz blockers are encountered during fuzz testing. Our system consists of a fuzz blocker localizer and an LLM-based agentic framework capable of generating seeds with feedback-driven refinement. Our system extends beyond traditional LLM-based input generation approaches, which narrow the scope to focus on LLM-generatable formats, such as JSON, XML, and source code. To enable the generation of various types of inputs, including binary files, we incorporate a Python-based and Kaitai Struct [21]-based seed generator. Additionally, we provide the LLM with reference seeds to facilitate the generation of inputs. To mitigate LLM hallucination, we employ coverage-based feedback, along with various auxiliary tools, enabling the system to flip specific branches in the target function reliably. These findings demonstrate the potential of our approach to enhance grey-box fuzzing by supporting the execution of previously unreachable paths in a semantic manner.

This study is part of the AIxCC competition; this capability is not only valuable for resolving fuzz blockers but also contributes to the broader AIxCC objective by enabling the generation of seed inputs that can reach suspicious code regions. If the existing seed pool is unable to reach a target function, our system can analyze the call chain, identify the blocking condition, and generate inputs to unlock the blocked path.

This report presents the development and technical validation of a system designed to identify and overcome fuzz blockers, facilitating deeper and more targeted software testing in large-scale, diverse open-source codebases.

2 Related Works

Fuzzing open-source software, specifically, OSS-Fuzz [23] has conducted cluster-level fuzzing on open-source projects and reported discovered crashes. The range of testing includes not only command-line programs, but also more complex systems such as HTTP servers [22], GUI software, and even device drivers. To fuzz such software using existing fuzzers [3, 15, 33], a corresponding fuzz driver or fuzz harness is required to forward random byte strings to the proper APIs. However, despite the novelty of greybox fuzzing, its heavy reliance on random mutations poses limitations, particularly in exploring code paths beyond a certain depth. As a result, deeply nested or complex execution paths often remain untested.

Recently, large language models (LLMs) have been developed and utilized in various areas. Large Language Models (LLMs) are a class of deep learning models constructed with carefully designed neural network architectures [26] and a vast number of parameters, enabling them to store and represent a large amount of information. These models are capable of identifying patterns from user-provided input prompts and generating appropriate responses [2]. Moreover, through Reinforcement Learning from Human Feedback(RLHF) [20], they are further trained to produce helpful and user-aligned responses. LLMs are typically trained to predict the next token given a sequence of input tokens. During generation, they follow an autoregressive decoding scheme: the model predicts the next token, appends it to the end of the sequence, and then uses it as input for the next token prediction. This process continues iteratively, producing coherent and context-aware outputs. Depending on the provided prompt, LLMs can interact with users in a conversational manner. Recently, advanced prompting techniques beyond simple dialogue have also been explored [1, 4, 27, 28, 31].

Moreover, LLMs have evolved in a "thinking mode", where they internally generate intermediate thoughts before producing a final answer [5, 6, 9, 10, 19]. These models often begin by generating internal representations that they consider useful for solving the given task, and use them as a precondition to produce more accurate and reliable responses. This approach typically utilizes Monte Carlo Tree Search, where the model is trained to select token sequences with higher expected accuracy. By simulating and evaluating multiple reasoning paths before answering, these models demonstrate improved performance on complex tasks requiring multistep reasoning.

From this background, there has been a growing interest in applying LLMs to fuzzing. These approaches may involve generating fuzz harnesses(drivers) or producing a seed corpus. The generation of fuzz drivers typically requires substantial domain-specific knowledge about the target library or API, making the process human-intensive. To alleviate this, several studies have proposed automating harness generation using LLMs [16, 23, 29]. OSS-Fuzz-Gen [23] attempts to generate a harness for each individual function, while PromptFuzz [16] explores interactions across multiple APIs. However, the harness generation presents a notable challenge: evaluating the correctness of the generated harness. When crashes are triggered, it is often difficult to reach a consensus on whether the crash is meaningful or realistically exploitable in real-world scenarios.

There have been numerous attempts to apply LLMs to fuzz corpus generation [7, 12, 17, 18, 25]. Most of these efforts have focused on LLM-generatable inputs, such as XML, JSON, or source code, often targeting specific domains. How-

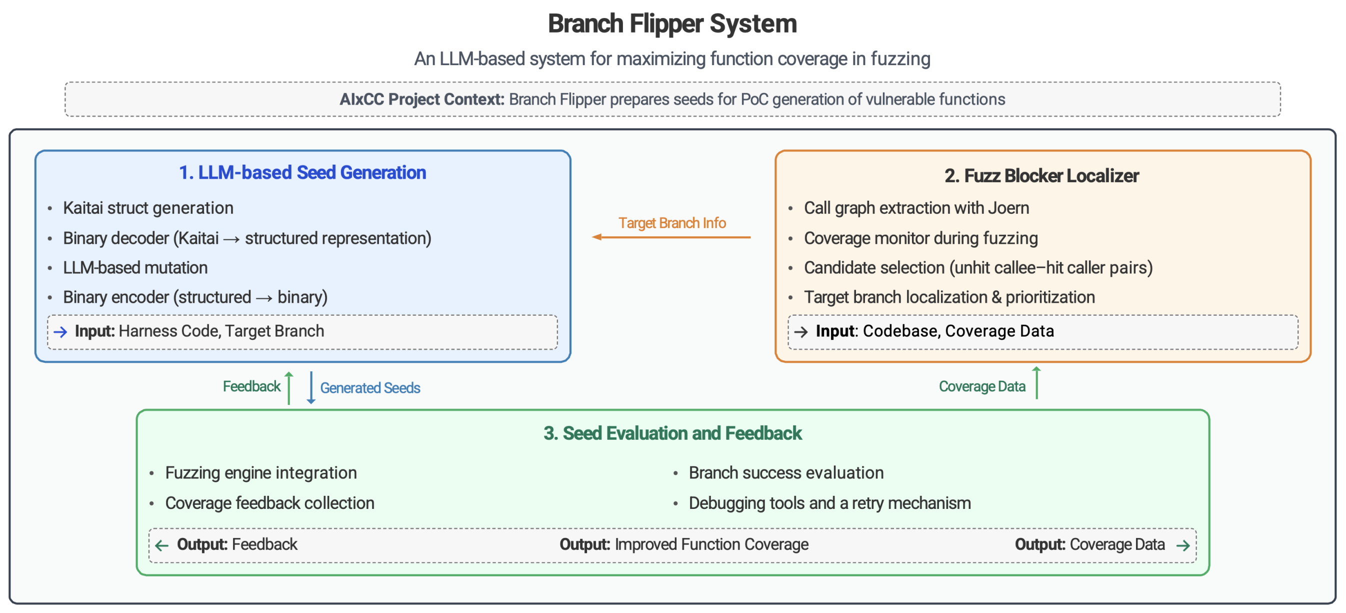

Branch Flipper System
An LLM-based system for maximizing function coverage in fuzzing
AIxCC Project Context: Branch Flipper prepares seeds for PoC generation of vulnerable functions
1. LLM-based Seed Generation
Kaitai struct generation
Binary decoder (Kaitai → structured representation)
LLM-based mutation
Binary encoder (structured → binary)
Input: Harness Code, Target Branch
Target Branch Info
2. Fuzz Blocker Localizer
Call graph extraction with Joern
Coverage monitor during fuzzing
Candidate selection (unhit callee–hit caller pairs)
Target branch localization & prioritization
Input: Codebase, Coverage Data
Feedback
Generated Seeds
Coverage Data
3. Seed Evaluation and Feedback
Fuzzing engine integration
Coverage feedback collection
Branch success evaluation
Debugging tools and a retry mechanism
Output: Feedback
Output: Improved Function Coverage
Output: Coverage Data

Figure 1: Overall architecture of the Branch Flipper. The system consists of three components: (1) a seed generator, (2) a branch target localizer, and (3) a seed evaluation and feedback module. Each component runs in a peer-to-peer manner, where each agent is self-directed, performing local tasks and initiating tool calls or requesting other agents via a message queue.

ever, applying these methods to binary-level software remains challenging. To address these limitations, some approaches generate Python scripts instead, which are then executed to produce binary inputs [24]. Nonetheless, due to the latency of LLM, fully replacing traditional mutation engines is infeasible in terms of throughput. As a result, a common strategy has augmented existing fuzzing workflows by generating a limited number of high-quality seed corpora, which can then be used to enhance or bootstrap the mutation-based fuzzing process [17].

Some studies have tried to fully automate penetration testing using LLM agents [32], which can plan, execute, and self-correct from feedback in a fully end-to-end manner [8, 13]. The AI Cyber Challenge (AIxCC), organized by the U.S. Defense Advanced Research Projects Agency (DARPA), is a gamified autonomous hacking competition. In the competition, teams are provided with a selection of open-source projects of various programming languages and domains in the OSS-Fuzz format. Teams then deploy AI agents that identify vulnerabilities and generate proof-of-concept (PoC) seed inputs without any human intervention, and submit patches for discovered bugs. Points are awarded when a submitted seed triggers a sanitizer when run through the predefined harness, indicating a successful vulnerability discovery. However, duplicate submissions that stem from the same root cause are penalized. Projects are presented in two modes: diff mode, which focuses on identifying vulnerabilities introduced in recent Git commits, or full mode, which analyzes the entire codebase without considering the editing history. After discovering vulnerabilities, teams can also submit patches, and points are granted if the patch prevents the vulnerability while maintaining its functionality.

3 Method

In this section, we introduce the Branch Flipper, an LLM-based seed generation for unlocking fuzz blockers. The Branch Flipper comprises several components: LLM-based seed generation using Kaitai Struct [21], a Joern [14]-based fuzz blocker localizer, LLM-based seed generation, and a coverage-based feedback loop.

3.1 Overview

This work, *Branch Flipper*, is part of the AIxCC project. Our AIxCC system runs fuzzers in the background with the Branch Flipper to prepare a collection of seeds, which serve as a baseline input when the system attempts to generate a PoC of corresponding vulnerable functions. Thus, to acquire at least one seed for each function, this work aims to maximize function coverage by generating seeds that unlock fuzz blockers and support the discovery of new functions.

Our proposed Branch Flipper system consists of three major components: (1) a seed generator, (2) a branch target localizer, and (3) a seed evaluation and feedback.

LLM-based seed generation has been an active area of research [7, 12, 17, 18, 24, 25], with most prior works focusing

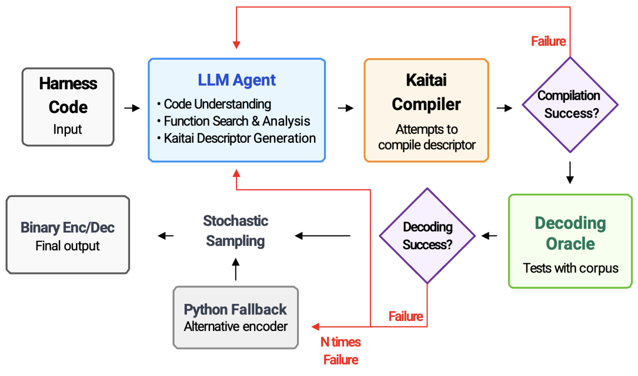


Figure 2: Workflow of a Kaitai descriptor generation.

on natural language-compatible formats, and directly generating binary inputs remains challenging. Some approaches [24] guide LLMs to generate Python scripts, which are then executed to produce binary files.

To support more flexible and LLM-friendly binary handling, we utilize Kaitai Struct [21], a structured YAML-like language for describing binary formats. Modern LLMs are typically familiar with this format, making it a practical choice for bridging the gap between natural language capability and binary manipulation. In our system, given a harness code, the LLM attempts to generate a corresponding Kaitai Struct descriptor. From a pool of Kaitai descriptors, the system selects one to decode the binary, yielding a structured representation, and then feeds it to an LLM. The LLM then mutates this structured representation based on the analysis of the targeted branch blocker, re-encodes it into a new binary input. If the targeted branch is successfully taken by the new input, the system proceeds to the next candidate; otherwise, it requests a retry with the aid of debugging tools.

To localize the fuzz blocker, we initially extract the function call graph using Joern [14] from the whole codebase. During fuzzing, we monitor function coverage, and if a function is covered but its callee remains unhit, these unhit callees and caller pairs are collected as candidates. The most impactful candidate then becomes the next target. Concurrently, we collect the seeds that reach each function and use them as a baseline for seed generation.

Through this iterative loop, we aim to generate blocker-unlocker with enhanced binary generation capabilities and grounding LLM hallucinations. Ultimately, this enables the LLM to contribute meaningfully to branch blockers within the greybox fuzzing pipeline.

3.2 LLM-based Seed Generation

3.2.1 Kaitai Struct Generation

Previous studies have primarily focused on natural language-compatible seed formats, those used in JSON/XML parsers or compilers, allowing them to directly generate fuzzing seeds and use them as-is during fuzzing. However, due to the inherent limitations of LLMs in directly constructing low-level byte streams, [24] has attempted to generate Python code that creates a binary file. This approach is particularly useful when fuzzing harnesses rely on FuzzedDataProvider-style APIs, which require a precise understanding and crafting of byte-level inputs.

Kaitai Struct [21] offers a YAML-based structured language for describing binary formats. Given a description, the Kaitai compiler generates Python modules that provide both parsers and generators for the corresponding byte stream. In our study, we support two methods for generating byte stream inputs:

1. The traditional approach of using LLMs to generate Python-based parsers and generators(decoders and encoders).
2. A new approach that leverages LLM-generated Kaitai Struct descriptors, which are then compiled and used to parse and generate binary inputs.

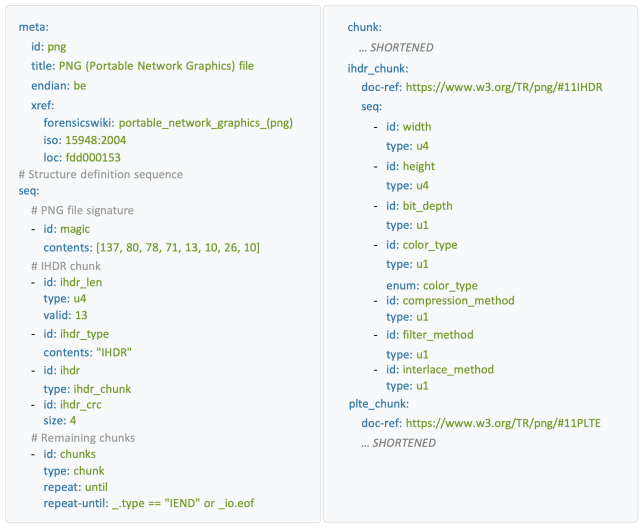
The system provides a fuzzing harness code to the LLM and instructs it to first understand the code. The LLM is capable of searching function definitions, references, and reading the code to understand the context. It then attempts to generate a Kaitai struct descriptor based on the understanding. If the Kaitai compiler fails to compile the descriptor, the system returns the compiler error to the LLM, prompting a revision until a valid and compilable Kaitai description is produced.

Modern LLMs such as GPT-4o, o4-mini, o3, and Claude 3.5 Sonnet already know and understand Kaitai Struct and often demonstrate notable generation capabilities for near-correct descriptors. However, syntactic and semantic errors also remain common. Hence, we construct an iterative feedback loop involving compilation success/failure signals to guide the LLM toward producing compilable Kaitai descriptors.

However, the performance of LLMs tends to degrade with longer context lengths [11]. Empirically, we observed that when the provided source code or conversation becomes lengthy, the compile success rate decreases. We also noticed that LLMs sometimes oversimplify Kaitai descriptions, omitting crucial structural details to minimize compile errors. Despite these issues, Kaitai-based representations can sometimes provide a more precise description of binary inputs than their Python-only counterparts.

In practice, to maximize the system's effectiveness, we adopt a stochastic strategy: for each seed generation task, the system stochastically chooses between the Python-based encoder and the Kaitai struct encoder. And also in cases where Kaitai compilation fails, the Python-based encoder serves as a fallback.

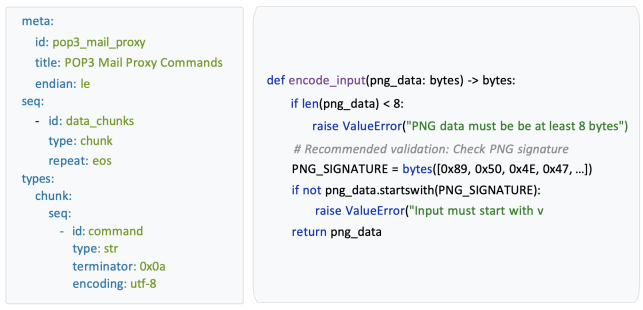
To be effective, the Kaitai descriptor must not only be compilable but also correctly decode actual seeds. Since our gen-



```
meta:
  id: png
  title: PNG (Portable Network Graphics) file
  endian: be
  xref:
    forensicswiki: portable_network_graphics_(png)
    iso: 15948:2004
    loc: fdd000153
# Structure definition sequence
seq:
  # PNG file signature
  - id: magic
    contents: [137, 80, 78, 71, 13, 10, 26, 10]
  # IHDR chunk
  - id: ihdr_len
    type: u4
    valid: 13
  - id: ihdr_type
    contents: "IHDR"
  - id: ihdr
    type: ihdr_chunk
  - id: ihdr_crc
    size: 4
  # Remaining chunks
  - id: chunks
    type: chunk
    repeat: until
    repeat-until: _.type == "IEND" or _io.eof
```

```
chunk:
  ... SHORTENED
ihdr_chunk:
  doc-ref: https://www.w3.org/TR/png/#11IHDR
  seq:
    - id: width
      type: u4
    - id: height
      type: u4
    - id: bit_depth
      type: u1
    - id: color_type
      type: u1
      enum: color_type
    - id: compression_method
      type: u1
    - id: filter_method
      type: u1
    - id: interlace_method
      type: u1
plte_chunk:
  doc-ref: https://www.w3.org/TR/png/#11PLTE
  ... SHORTENED
```

Figure 3: A sample descriptor generated by the Claude 3.5 Sonnet when the harness code of the project *libpng* is given.



```
meta:
  id: pop3_mail_proxy
  title: POP3 Mail Proxy Commands
  endian: le
seq:
  - id: data_chunks
    type: chunk
    repeat: eos
types:
  chunk:
    seq:
      - id: command
        type: str
        terminator: 0x0a
        encoding: utf-8
```

```
def encode_input(png_data: bytes) -> bytes:
    if len(png_data) < 8:
        raise ValueError("PNG data must be be at least 8 bytes")
    # Recommended validation: Check PNG signature
    PNG_SIGNATURE = bytes([0x89, 0x50, 0x4E, 0x47, ...])
    if not png_data.startswith(PNG_SIGNATURE):
        raise ValueError("Input must start with v
    return png_data
```

Figure 4: Some cases of oversimplified generated encoders of Kaitai and Python settings. Left one is generated Kaitai descriptor when the Nginx harness is given, and the right one is generated Python-based encoder when the libpng harness is given (the same harness as Figure 3.) Both are oversimplified and simply forward inputs to outputs as is.

erated decoders operate on the corpus generated by the fuzzer, they must be robust enough to handle a wide range of inputs. Therefore, we incorporate a decoding test (or oracle) after compilation: given a fuzzing corpus sample, the compiled decoder tries to decode it, and decoding failures are reported back to the LLM. This additional feedback helps guide the LLM to produce more resilient and generalizable Kaitai structures.

3.2.2 Seed Generation

Once a suitable encoder has been constructed (either Python-based or Kaitai-based), the LLM is now able to generate concrete binary seeds.

Our goal is to analyze a specific branch predicate and generate a suitable seed. However, reaching this particular predicate often requires passing through multiple prior control flows, and each imposes its own constraints on the inputs. As the depth of nested predicates increases, attempting to satisfy the target predicate from scratch (i.e., without initial guidance or baseline) while preserving all prior becomes exponentially more complex, especially given the context-length limitations of LLMs.

To address this, we utilize the collected seeds from the fuzzer run. We assume that we already possess a seed that has successfully reached the function containing the target predicate. Using this as a starting point allows us to focus only on solving the target predicate, which has already satisfied the earlier branches, and we can effectively reduce the problem complexity.

In our system, given a seed that reaches the function containing the target branch predicate, we perform the following sequence:

1. The LLM is provided with the line number of the target branch, and the corresponding seed previously reached the function.

2. The LLM analyzes the branch predicate at the target location and uses the given seed as a starting point.

3. The LLM tries to generate a new seed that seems to take the desired branch.

Since raw binary seeds are uninformative to LLMs, we decode the seed into a structured JSON format using the previously generated decoders. This JSON representation allows the LLM to understand and reason about the input fields in a semantically meaningful way.

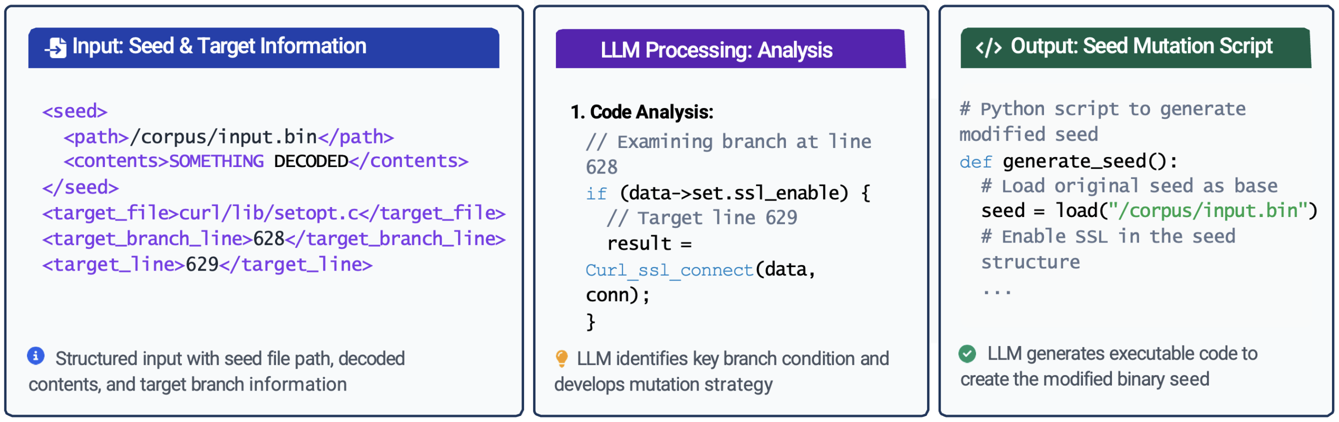
Based on this plan, the LLM generates a Python script that produces a binary seed aiming to satisfy the branch condition. This script is executed within a sandboxed environment to produce the final seed file. The resulting binary input is then executed within the fuzzing harness to verify whether the target branch was taken.

3.3 Fuzz Blocker Localizer

Our second challenge was how to localize fuzz blockers and feed them into the branch flipper. The current requirement is that unlocking a predicate within a given function must allow the discovery of previously unexplored functions. We addressed this in two stages: the first defines the frontier, and the second localizes the targeting branch within that frontier.

3.3.1 Call Graph and the Frontiers

The first stage involves defining the *frontier*. A frontier refers to a function candidate that has been hit at least once during fuzzing and has at least one callee function that has never been invoked. In other words, if a certain predicate in this


Input: Seed & Target Information

```
<seed>
  <path>/corpus/input.bin</path>
  <contents>SOMETHING DECODED</contents>
</seed>
<target_file>curl/lib/setopt.c</target_file>
<target_branch_line>628</target_branch_line>
<target_line>629</target_line>
```

Structured input with seed file path, decoded contents, and target branch information
LLM Processing: Analysis
1. Code Analysis:

```
// Examining branch at line
628
if (data->set.ssl_enable) {
  // Target line 629
  result =
Curl_ssl_connect(data,
conn);
}
```

LLM identifies key branch condition and develops mutation strategy
Output: Seed Mutation Script

```
# Python script to generate
modified seed
def generate_seed():
  # Load original seed as base
  seed = load("/corpus/input.bin")
  # Enable SSL in the seed
  structure
  ...
```

LLM generates executable code to create the modified binary seed

Figure 5: Workflow of the blocker-unlocker generation. The LLM is provided with structured inputs, including a decoded baseline seed, target file, and branch. Then, the LLM starts to analyze the predicate using certain tools, finding function definitions and references, reading them, and attempting to generate a new seed.

function is triggered, it could lead to the exploration of a new function that had not been visited before.

There are various ways to build a call graph. We employ Joern [14], a general-purpose static analyzer known for its speed and stability. Joern supports the extraction of a Code Property Graph (CPG), including call relationships. In C/C++ open-source projects, macro variables and compiler preprocessor often conditionally enable or disable certain code paths. As a result, call relationships visible in source code may not exist in the final compiled binary, creating false positives. Joern mitigates this by explicitly handling macros and automatically including relevant header files, yielding more reliable results than tools like GNU Global [30].

In parallel to the LLM Agent, we continuously run fuzzers in the background. Whenever the LLM generates a new seed, it is added to the fuzzer's seed corpus. This setup allows the system to continually integrate new seeds and expand function coverage through basic mutation. When a new seed is generated, a background process monitors this event, checks coverage, and updates the metadata from the call graph accordingly.

Each node in the call graph stores metadata, including its shallowest-known caller, the call depth, and a visit flag. When a new seed expands function coverage, we traverse the call graph, updating the caller metadata and propagating to its callee nodes. If a function node has at least one unvisited callee, it is labeled as a *frontier* and queued for branch flipping.

However, the number of such frontiers can be massive, often exceeding 10,000 per hour during early fuzzing. Invoking the LLM Agent for each of these is impractical in terms of computational budgets. In practice, some of these frontiers are unlocked by random mutators before being conveyed to the branch flipper agent. Therefore, we implement a hard waiting time for each queued frontier. Only after this waiting period, we check whether the callee remains unvisited, and only then proceed to branch flipping.

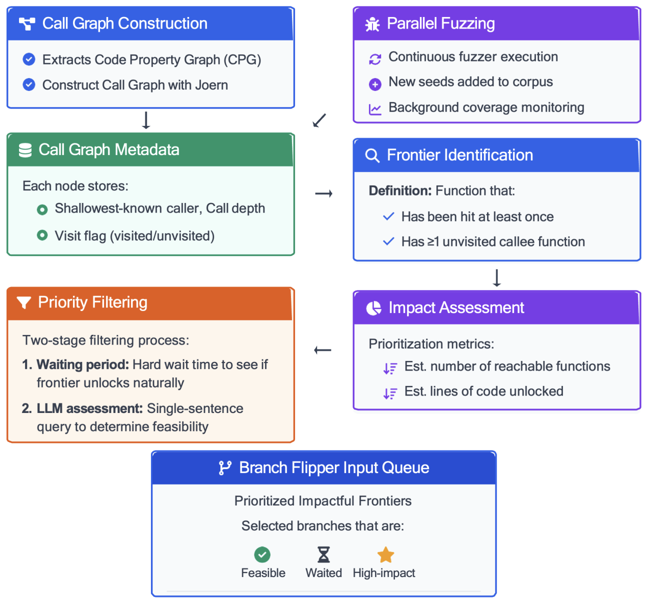
3.3.2 Branch Localization

Once a frontier is determined, we must identify a branch predicate to target. Since our goal is to trigger an unvisited function, we target branches that may call such functions.

Joern supports queries related to control structures. Given a call expression to an unvisited callee within a hit function, we can use Joern's CPG to locate the branch predicate enclosing the call. The first such branch is guaranteed to be un-taken. However, in nested control structures, this may not be the closest former branch predicate that has been evaluated, and identifying the correct one may require line-level coverage analysis. This can be costly due to complications like comments and parsing behavior. Given AIxCC's multi-language scope, we conduct a simplified strategy: we target only the immediate branch predicate preceding the call expression. Despite this simplification, we still observe a significant number of viable cases, and we believe flipping such branches remains highly meaningful. We left a more sophisticated localization of the earliest evaluated branch predicate as future work.

Even with precise localization, some branches may still be unreachable due to environmental constraints. For example, when a harness hardcodes certain variables, such as enabling logging, callee functions cannot be invoked by such variables.

To address this, we add a single-stage query to the LLM, issuing a one-sentence prompt to assess whether the branch is reachable. We only attempt flipping for those deemed feasible. Although this may yield false negatives, this filtering is acceptable since the number of candidate frontiers still exceeds our processing budget. Additionally, we prioritize candidate frontiers based on the estimated number of reachable functions or lines of code that can be explored upon unlocking



Call Graph Construction
Extracts Code Property Graph (CPG)
Construct Call Graph with Joern
Parallel Fuzzing
Continuous fuzzer execution
New seeds added to corpus
Background coverage monitoring
Call Graph Metadata
Each node stores:
Shallowest-known caller, Call depth
Visit flag (visited/unvisited)
Frontier Identification
Definition: Function that:
Has been hit at least once
Has ≥1 unvisited callee function
Priority Filtering
Two-stage filtering process:
1. Waiting period: Hard wait time to see if frontier unlocks naturally
2. LLM assessment: Single-sentence query to determine feasibility
Impact Assessment
Prioritization metrics:
Est. number of reachable functions
Est. lines of code unlocked
Branch Flipper Input Queue
Prioritized Impactful Frontiers
Selected branches that are:
Feasible
Waited
High-impact

Figure 6: Workflow from collecting frontiers to transporting to the branch flipper agent. Frontiers are collected by the background coverage monitor with a call graph and assessed for their impact through the estimated reachable functions that have not been explored before.

the blocker, thereby guiding the flipper toward high-impact branches, which we previously referred to as the *impactful frontier*.

3.4 Seed Evaluation and Feedback

After the target branch predicate has been localized and the LLM generates a corresponding mutated seed, we perform a grounding step to evaluate the effectiveness of the attempt.

Once a new seed is generated, we execute the fuzz harness with it and collect coverage information. If the target function is reached or the unhit target branch is taken, the seed is marked as a success and proceeds to the next branch candidates. However, if coverage does not reflect success, it is crucial to inform the LLM about the progress made in execution.

Specifically, we compute the nearest hit line before and after the target branch and feed this information back to the LLM. This allows the agent to estimate which portions of the program were executed and where progress was stuck. The LLM then re-analyzes the code and attempts a new seed generation along with some debugging trials.

However, we observe that even after multiple failures, the LLM may hallucinate success, claiming that the predicate was likely satisfied and that coverage measurement is faulty. In these cases, the LLM often continues to generate seeds under false assumptions.

To counteract this, we enforce a strict grounding. If coverage data indicates that the target was not reached, the system requests at least one additional retry, regardless of the LLM's confidence. By introducing retry policies, we improve the reliability and stability.

3.5 Further Details

Unlike centralized architectures where a single orchestrator governs all tool calls, our system adopts a peer-to-peer agentic model, in which LLM agents with distinct objectives communicate through message queue interfaces. Each agent is self-directed, performing localized reasoning tasks and initiating tool calls or requests to other agents as necessary.

The background fuzzer continuously monitors the evolution of the seed corpus. Upon detecting new frontiers, it triggers a localization routine and stacks it as a candidate. The seed generator pulls a candidate and tries to generate a new seed. It can request coverage verification from the background fuzzer, forming a mutually dependent workflow. This decentralized architecture removes the bottleneck of a single orchestrator and supports a more scalable, fault-tolerant execution flow.

The other challenge is managing context length, especially when operating on low-level codebases, such as those written in C. In these settings, individual functions often span dozens of lines, and naively injecting full source code can rapidly exhaust the context length, degrading performance [11].

To address this, we implement a dedicated Source Review Agent. When an LLM agent requires a detailed understanding of specific code regions (e.g., to infer predicate conditions or data structures), it queries the Source Review Agent instead of embedding raw code into its own prompt. This subagent handles code lookup and semantic summarization. Thus, the parent agent can operate within a compact and structured representation of the source, retaining high-fidelity analysis.

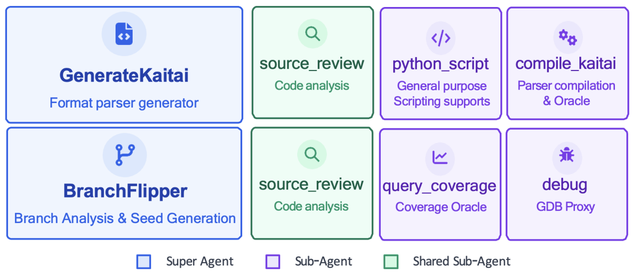


Figure 7: LLM Agents and their corresponding tools.

In practice, we also observe LLM's tool usage biases, where only a few tools are consistently invoked. To make the system more consistent and compact, initially, we added all potentially useful tools and observed the tool call tendencies. We ensure that frequently used tools execute before invoking the agent and inject the execution result into the agent prompt, if possible. For example, initially, we gave a path to the harness, and we observed that the agent always read the harness first. We added the harness code to the prompt and removed the harness reading tools. On the other hand, rarely used or optional tools are either moved to subagents or completely removed. This strategy reduces tool overload while ensuring necessary data is available without active tool invocation.

We structure the overall execution into a tree of LLM subagents, each responsible for a narrow task domain. Upon the system initialization, a target project is loaded, and Kaitai and Python-based encoders and decoders are generated and stored in a centralized database. During fuzzing, when a frontier is detected, the localizer agent identifies the corresponding branches and saves them to a candidate database. The branch flipper pulls the most impactful candidate and tries to generate a seed. The flipper receives the harness code, a stochastically chosen encoder, decoder, and the definition of the target function, as well as the branch location and a visualized version of the input seed. It solves subtasks using three primary tools: the Source Review Agent, the Debugger Agent (which receives high-level debugging instructions and operates GDB to gather information), and the Coverage Query Agent.

4 Experiments

We conducted experiments on several projects, targeting C/C++-based applications such as Curl, Nginx, libpng, and Zstd, as well as the Java-based application Tomcat. These targets include both benchmark applications provided by AIxCC and external open-source projects. In the competition, we utilized various LLMs according to their strengths for several tasks, and the experiments presented below were conducted using Claude 3.5 Sonnet.

Generating Kaitai Descriptor: The first step is generating Kaitai descriptors. The LLM Agent is composed of tools such as source_question, python_script, and compile_kaitai to produce the Kaitai descriptor. For the Python-based encoder, the compile_kaitai tool is replaced with a sanity_check tool, which also performs oracle-like checks using the existing corpus. For each project, we selected harnesses and attempted to generate both encoders and decoders.

To evaluate the generated Kaitai Structs, we manually wrote Kaitai descriptors as ground-truth for 11 harnesses, either provided by the AIxCC competition organizers or collected by ourselves for each project. We then defined the following evaluation criteria:

1. **Description precision**: The proportion of declared elements in the ground-truth structure that were successfully covered.
2. **Encoder precision**: The proportion of cases where the encoded byte stream exactly matched the original input stream.

Harness	Baseline	Kaitai
Identity Mapping	*100%*	*100%*
curl: fuzz_url.cc	100%	100%
zstd: block_decompress.c	100%	100%
Join-operation Only	*100%*	*100%*
nginx: fuzz/mail_request_harness.cc	100%	100%
nginx: fuzz/smtp_harness.cc	100%	100%
General Cases	*55.33%*	***84.66%***
curl: curl_fuzzer.cc	50%	**75%**
libpng: read_fuzzer.cc	16%	**100%**
nginx: pov_harness.cc	100%	100%
tomcat: DefaultServletFuzzer.java	66%	**100%**
tomcat: JNDIRealmFuzzer.java	100%	100%
zstd: stream_round_trip.c	0%	**33%**
Complex Case	*0%*	*0%*
zstd: sequence_compression_api.c	0%	0%
Total	*73.2%*	***90.8%***

Table 1: Top-3 Description precision improvement on each category. Italic font to the left of each category represents the average within the category, while the total at the bottom indicates the overall average. Bold font highlights cases where Kaitai Struct shows improvement over the Python encoder, demonstrating an overall positive trend.

3. **Decoder precision**: The proportion of fields correctly extracted from the input byte stream among all fields declared in the decoder's structure.

Among these, encoder and decoder precision are guaranteed by the compiler for Kaitai Struct; therefore, we focused on evaluating description precision. We treated each field in the declared descriptor as a single element and calculated the proportion by counting the fields whose endianness, type, length, and offset exactly matched those in the ground-truth, relative to the total number of fields.

In the Table 1, the baseline counts fields mentioned in the Python-based encoder where the corresponding encoding operation correctly packs the field's length, type, offset, and endianness. This corresponds to the previously defined encoder precision; since the Python-based encoder/decoder does not explicitly specify field information, this serves as a proxy for description precision. For Kaitai Struct, only clearly specified fields declared in the YAML specification are counted.

As shown in the result, for harness encoders requiring only identity mapping and join operations, the existing Python-based encoder has already achieved sufficiently meaningful results. However, in cases involving nested fields, Kaitai Struct demonstrated improved description precision. The Zstd cases were too complex cases where the incoming byte stream is treated as a pseudo-random seed to drive a random number generator, and the generated values are used as API arguments. It makes an encoder practically impossible to embed semantic information as API inputs. Considering this, Kaitai Struct

ID	Baseline	Kaitai
curl#1	Pass (0.17$, 7min)	Pass (0.70$, 8min)
curl#2	Pass (0.42$, 13min)	Pass (1.38$, 12min)
libpng#1	Pass (0.25$, 4min)	Pass (0.40$, 4min)
libpng#2	Pass (0.22$, 3min)	Pass (0.37$, 3min)
libpng#3	Pass (0.27$, 4min)	Pass (0.90$, 8min)
nginx#1	Fail (2.67$, 24min)	**Pass (0.64$, 11min)**
nginx#2	Pass (0.24$, 3min)	Pass (0.74$, 6min)
nginx#3	Pass (0.31$, 3min)	Pass (0.99$, 12min)
nginx#4	Pass (0.27$, 2min)	*Fail (2.58$, 19min)*
nginx#5	Pass (1.62$, 16min)	Pass (0.87$, 7min)
nginx#6	Fail (2.48$, 26min)	Fail (1.97$, 15min)
nginx#7	Pass (0.26$, 2min)	Pass (0.54$, 5min)
tomcat#1	Fail (2.02$, 30min)	**Pass (0.90$, 8min)**
tomcat#2	Fail (0.85$, 17min)	Fail (0.73$, 17min)
tomcat#3	Fail (2.07$, 30min)	Fail (1.34$, 20min)
zstd#1	Pass (0.15$, 2min)	Pass (0.52$, 5min)
zstd#2	Fail (1.90$, 20min)	Fail (1.40$, 11min)
zstd#3	Pass (0.31$, 4min)	*Fail (1.99$, 20min)*
Total	67% (17$, 210min)	67% (19$, 191min)

Table 2: Trials of PoC generation for given vulnerability with several variants. **ID** represents each vulnerabilities, **Baseline** use Python encoder only when generate a binary seed, **Kaitai** utilizes Kaitai-struct only.

generally showed meaningful accuracy across most cases.

In some cases, as seen in the Figure 3, 4, while the Python-based encoders often missed details like the PNG header and just formed an uninformative identity function, such details are clearly represented in the generated Kaitai descriptors. It seems that the LLMs already know the published descriptors and utilize them. For targets like Tomcat, where FuzzedDataProvider is used, we found that byte stream issues, such as endianness, were more accurately handled after integrating Kaitai. Especially in the case of Zstd, which has a highly complex FuzzedDataProvider logic, the original Python encoder is often reduced to trivial identity mappings. In contrast, the LLM was encouraged to generate more sophisticated byte descriptions when utilizing Kaitai-struct, even if partially.

We then used these encoders to attempt PoC seed generation, in the Table 2, as part of the AIxCC competition. During seed generation, we observed that it successfully triggers vulnerabilities with the generated PoC when using Kaitai descriptors. Since these results were complementary to the ones generated with Python-based encoders, this provides indirect evidence that using both in a dual-path setup is a meaningful strategy for branch flippers.

Branch Flipper: Next, we moved on to exploring branch blockers and executing the flipper. Before we built an algorithmic localizer, we had the LLM directly localize the targeted branches. However, case studies revealed that more than half of the attempts failed to localize the branches correctly. Algorithmic localization, on the other hand, significantly reduced such mislocalizations.

Project	Samples	Flipped
curl	13	7 (53.8%)
nginx	11	3 (27.3%)
tomcat	10	7 (70.0%)
PoC Generation	13	9 (69.2%)
Total	47	26 (55.3%)

Table 3: The success rates of precollected branch blockers. The three projects listed under **Project** represent high-priority solvable blockers identified during actual fuzzing, while the "PoC Generation" category refers to blockers collected from cases where the target was unreachable during PoC generation.

Project	Trials	Flipped	Mark
java: compress	27	3(11.11%)	4(14.81%)
c: curl	10	2(20%)	2(20%)
c: dropbear	6	1(16.67%)	
c: freerdp	10	6(60%)	
c: libexif	2	0(0%)	
c: libpng	27	6(22.22%)	
c: libxml2	6	1(16.67%)	
c: nginx	7	2(28.57%)	1(14.28%)
c: sqlite3	1	0(0%)	1(100%)
java: tika	3	0(0%)	
java: zookeeper	15	0(0%)	2(13.33%)
c: zstd	2	0(0%)	
Total	116	21(18.10%)	10(8.62%)

Table 4: 24-hour running results of Branch Flipper, while running LibFuzzer concurrently with the public OSS-Fuzz corpus seeded. **Trials** indicates the number of branch flipping attempts, **Flipped** for cases mechanically judged as successful, and **Mark** for mechanically judged as failures but deemed successful by the LLM.

We collected branch blocker cases that were ensured to be solvable for each harness and attempted to flip them. Since we intended to use Branch Flipper for both fuzz blocker resolution and PoC generation, we sampled from two groups: those selected as high priority by the blocker localizer during actual fuzzing, and those PoC generation failures where the target functions were not reached.

Among the branch blockers on the Table 3, some were semantically trivial, and the branch flipper was able to solve those cases without difficulty. However, flipping failed in some cases despite being considered solvable. Examining the failure cases, we found that when the LLM performed more than 10 turns of flipping, it often exhausted the maximum allowed turns by repeating past failed attempts without making any meaningful new trials. Even in early stages, if the preconditions of the preceding branches were too complex, the

ID	Baseline	+ Branch Flipper
curl#1	Pass (0.17$, 7min)	Pass (0.12$, 11min)
curl#2	Pass (0.42$, 13min)	Pass (0.44$, 14min)
libpng#1	Pass (0.25$, 4min)	Pass (0.18$, 5min)
libpng#2	Pass (0.22$, 3min)	Pass (0.16$, 3min)
libpng#3	Pass (0.27$, 4min)	Pass (0.35$, 9min)
nginx#1	Fail (2.67$, 24min)	**Pass (0.21$, 4min)**
nginx#2	Pass (0.24$, 3min)	Pass (0.25$, 6min)
nginx#3	Pass (0.31$, 3min)	Pass (0.31$, 8min)
nginx#4	Pass (0.27$, 2min)	*Fail (1.50$, 15min)*
nginx#5	Pass (1.62$, 16min)	Pass (0.27$, 4min)
nginx#6	Fail (2.48$, 26min)	**Pass (0.80$, 11min)**
nginx#7	Pass (0.26$, 2min)	Pass (0.22$, 4min)
tomcat#1	Fail (2.02$, 30min)	**Pass (0.36$, 15min)**
tomcat#2	Fail (0.85$, 17min)	**Pass (1.65$, 46min)**
tomcat#3	Fail (2.07$, 30min)	Fail (1.36$, 23min)
zstd#1	Pass (0.15$, 2min)	Pass (0.11$, 3min)
zstd#1	Fail (1.90$, 20min)	Fail (1.35$, 13min)
zstd#3	Pass (0.31$, 4min)	Pass (0.81$, 8min)
Total	67% (17$, 210min)	**83% (11$, 202min)**

Table 5: Trials of PoC generation for given vulnerability with and without Branch Flipper. **ID** represents each vulnerabilities, **Baseline** without branch flipper and **+ Branch Flipper** with branch flipper when generating PoC.

model sometimes failed to consider preconditions, resulting in the execution not reaching the caller function after mutation.

While such issues might be resolved by employing advanced reasoning LLMs like o3 or some "thinking mode" models, the branch flipping requests often exceed tens of thousands per hour, resulting in significant costs and a burden on computational budgets. On the other hand, it means that each project accumulates over 10,000 branch blockers per hour, and solving even just the not-that-difficult cases with cheaper LLMs can make a reasonable and practical complement to random mutation-based greybox fuzzing.

We also ran the complete Branch Flipper system over a single day, from the public OSS-Fuzz corpus seeded. As shown on the Table 4, a total of 116 branch flipping trials were triggered, and each agent trial involved up to 30 turns of interaction, resulting in 21 successful branch flips (18.10%). Most successful cases achieved flipping within three iterations of mutation. Among the failure cases, some were mechanically misjudged due to discrepancies between lines localized by Joern and coverage data obtained from the fuzzers. In such cases, the LLM marked the attempt as successful, and these were separately logged as *marked by LLMs*. We estimate the Branch Flipper's success rate to be approximately between 18.10% and 26.72%. This indicates that the system not only provides expectable support during short fuzzing periods but also offers further benefit even when a corpus is already seeded.

We also investigated the applicability of our method to PoC generation. In practice, we observed that while PoC generators could often identify suspicious or vulnerable code regions, they failed to produce seeds that could successfully exploit them. A common cause was the inability to reach the target functions. To address this, we integrated our branch flipper tool into the PoC generation pipeline. Experimental results on Table 5 showed that our tool enabled successful PoC generation in cases where the original system had previously failed due to unreachable functions, improving the success rate from 66.67% to 83.33% while also reducing the cost of both LLM usage and execution time. Moreover, beyond simply reaching the target functions, the branch flipper enabled the process to be decoupled into two stages: generating inputs that reach the vulnerable function and then modifying them to become exploitable, resulting in improved overall performance in some cases.

5 Conclusion

We propose a cooperative agentic system designed to unlock fuzz blockers in a greybox fuzzing environment. The system operates with the goal of achieving coverage over previously unhit functions by localizing blockers and generating targeted seed inputs that can take the obstructing branches.

To support this, we suggest using an LLM to generate a binary encoder and decoder, utilizing a Python sandbox and the Kaitai-struct compiler. The LLM leverages subagents to analyze the execution path, such as a source code review agent and a GDB-based debugging agent. Based on reference seeds that partially reach the target region, the LLM generates new candidate seeds and evaluates them iteratively, incorporating coverage-based grounding to mitigate hallucinations.

Using this system, we successfully unlocked real-world fuzz blockers, contributing as one of the components in the AIxCC challenge pipeline. However, we observed that many blockers remain unsolvable, and in such cases, continued operation of the LLM-based agent can lead to inefficient use of computational resources and budget. This highlights a next step of future work: developing a method to predict the solvability of fuzz blockers, thereby improving resource efficiency in autonomous fuzzing workflows.

References

[1] Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., and Hoefler, T. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence 38*, 16 (Mar. 2024), 17682–17690.

[2] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *Advances in Neural Information Processing Systems* (2020), H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., pp. 1877–1901.

[3] Chen, P., Xie, Y., Lyu, Y., Wang, Y., and Chen, H. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2023), CCS '23, Association for Computing Machinery, p. 1600–1614.

[4] Chen, X., Pun, C.-M., and Wang, S. Medprompt: Cross-modal prompting for multi-task medical image translation, 2023.

[5] Claude. Claude 3.7 sonnet and claude code.

[6] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[7] Deng, Y., Xia, C. S., Peng, H., Yang, C., and Zhang, L. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023), ISSTA 2023, Association for Computing Machinery, p. 423–435.

[8] Fang, R., Bindu, R., Gupta, A., Zhan, Q., and Kang, D. Llm agents can autonomously hack websites, 2024.

[9] Google. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

[10] Guan, X., Zhang, L. L., Liu, Y., Shang, N., Sun, Y., Zhu, Y., Yang, F., and Yang, M. rstar-math: Small llms can master math reasoning with self-evolved deep thinking, 2025.

[11] Hsieh, C.-P., Sun, S., Kriman, S., Acharya, S., Rekesh, D., Jia, F., Zhang, Y., and Ginsburg, B. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654* (2024).

[12] Hu, J., Zhang, Q., and Yin, H. Augmenting greybox fuzzing with generative ai, 2023.

[13] Isozaki, I., Shrestha, M., Console, R., and Kim, E. Towards automated penetration testing: Introducing llm benchmark, analysis, and improvements. In *Adjunct Proceedings of the 33rd ACM Conference on User Modeling, Adaptation and Personalization* (New York, NY, USA, 2025), UMAP Adjunct '25, Association for Computing Machinery, p. 404–419.

[14] joern.io. Joern: The Bug Hunter's Workbench, Jan. 2024.

[15] LLVM. libfuzzer – a library for coverage-guided fuzz testing.

[16] Lyu, Y., Xie, Y., Chen, P., and Chen, H. Prompt fuzzing for fuzz driver generation, 2024.

[17] Meng, R., Duck, G. J., and Roychoudhury, A. Large language model assisted hybrid fuzzing, 2024.

[18] Meng, R., Mirchev, M., Böhme, M., and Roychoudhury, A. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)* (2024).

[19] OpenAI. Introducing o3 and o4-mini.

[20] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P. F., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems* (2022), S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., pp. 27730–27744.

[21] Project, K. Kaitai struct: declarative binary format parsing language, 2025.

[22] Reese, W. Nginx: the high-performance web server and reverse proxy. *Linux J. 2008*, 173 (Sept. 2008).

[23] Serebryany, K. OSS-Fuzz - google's continuous fuzzing service for open source software. USENIX Association.

[24] Shi, W., Zhang, Y., Xing, X., and Xu, J. Harnessing large language models for seed generation in greybox fuzzing, 11 2024.

[25] Shou, C., Liu, J., Lu, D., and Sen, K. Llm4fuzz: Guided fuzzing of smart contracts with large language models, 2024.

[26] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc.

[27] Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations* (2023).

[28] Wei, J., Wang, X., Schuurmans, D., Bosma, M., ichter, b., Xia, F., Chi, E., Le, Q. V., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems* (2022), S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., pp. 24824–24837.

[29] Xia, C. S., Paltenghi, M., Le Tian, J., Pradel, M., and Zhang, L. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Apr. 2024), ICSE '24, ACM, p. 1–13.

[30] Yamaguchi, S. Gnu global.

[31] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems* (2023), A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36, Curran Associates, Inc., pp. 11809–11822.

[32] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations* (2023).

[33] Zalewski, M. American fuzzy lop.