


OverHAuL: Harnessing Automation for C Libraries with Large Language Models

BSc Thesis

Konstantinos Chousos 
sdi2000215@di.uoa.gr

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens

July, 2025

Software vulnerabilities remain pervasive and challenging to detect, making robust testing approaches imperative. Fuzzing is an established software testing method for uncovering such vulnerabilities, through random input execution. Recent research has leveraged Large Language Models (LLMs) to enhance fuzz driver generation. However, most contemporary tools rely on additional resources beyond the target code, such as client programs or preexisting harnesses, limiting their scalability and applicability. In this thesis, we present OverHAuL, a neurosymbolic AI system that employs LLM agents to automatically generate fuzzing harnesses directly from library code, eliminating the need for auxiliary artifacts. To comprehensively evaluate OverHAuL, we construct a benchmark suite consisting of ten open-source C libraries. Our empirical analysis demonstrates that OverHAuL achieves an 81.25% success rate in harness generation across the evaluated projects, underscoring its effectiveness and potential to facilitate more efficient vulnerability discovery.

Preface

This thesis was prepared in Athens, Greece, during the academic year 2024–2025, fulfilling a requirement for the Bachelor of Science degree at the [Department of Informatics and Telecommunications](#) of the [National and Kapodistrian University of Athens](#). The research presented herein was carried out under the supervision of Prof. [Thanassis Avgerinos](#) and in accordance with the guidelines stipulated by the department. All processes and methodologies adopted during the research adhere to the academic and ethical standards of the university. The final version of this thesis is [hosted online](#) and is also archived in the department’s records, made publicly accessible through the university’s digital repository [Pergamos](#).

*To my beloved parents who, through their example, taught me patience, resilience and
perseverance.*

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Thanassis Avgerinos, for his insightful guidance, patience, and unwavering encouragement throughout this journey. His openness and our shared passion for the subject greatly enhanced my enjoyment of the thesis process.

I am also thankful to my fellow group members in Prof. Avgerinos' weekly meetings, whose willingness to exchange ideas and offer support was invaluable. My appreciation extends to Jorgen and Phaeton, friends who provided thoughtful input and advice along the way.

A special *thank you* goes to my parents Giannis and Gianna, Christina, and my friends for their constant support and understanding. Their patience and encouragement helped me persevere through this challenging period.

Table of contents

1. Introduction	1
1.1. Thesis Structure	2
1.2. Summary of Contributions	2
2. Background	3
2.1. Fuzz Testing	3
2.1.1. Motivation	5
2.1.2. Methodology	6
2.1.3. Challenges in Adoption	8
2.2. Large Language Models	8
2.2.1. State-of-the-art GPTs	9
2.2.2. Prompting	9
2.2.3. LLMs for Coding	10
2.2.4. LLMs for Fuzzing	11
2.3. Neurosymbolic AI	12
3. OverHAuL’s Design	13
3.1. Installation and Usage	14
3.2. Architecture	15
3.2.1. Project Analysis	15
3.2.2. Harness Creation	16
3.2.3. Harness Evaluation	16
3.3. OverHAuL Techniques	17
3.3.1. Feedback Loop	17
3.3.2. React Agents Triplet	17
3.3.3. Codebase Oracle	18
3.4. High-Level Algorithm	19
3.5. Scope	20
3.6. Implementation	20
3.6.1. Development Tools	21
3.6.2. Reproducibility	21
4. Evaluation	29
4.1. Experimental Benchmark	29
4.1.1. Local Benchmarking	30

4.2.	Results	31
4.2.1.	RQ 1: Can OverHAuL generate working harnesses for unfuzzed C projects?	33
4.2.2.	RQ2: What characteristics do these harnesses have? Are they similar to man-made harnesses?	33
4.2.3.	RQ3: How do LLM usage patterns influence the generated harnesses?	33
4.2.4.	RQ4: How do different symbolic techniques affect the generated harnesses?	35
4.3.	Discussion	35
4.3.1.	Threats to Validity	35
5.	Related work	37
5.1.	Static and Dynamic Analysis-Powered Fuzzing	37
5.2.	Extra Resources Required	39
5.3.	Only Source Code Required	40
5.4.	Differences With OverHAuL	41
6.	Future Work	43
6.1.	Enhancements to Core Features	43
6.2.	Experimentation with Large Language Models and Data Representation	44
6.3.	Comprehensive Evaluation and Benchmarking	44
6.4.	Practical Deployment and Community Engagement	45
7.	Conclusion	46
	Bibliography	48
	Appendices	56
A.	Abandoned Techniques	56
B.	Sample Generated Harnesses	58
B.1.	clibs/buffer	58
B.2.	willemt/cbuffer	60
B.3.	dvhar/dateparse	63
B.4.	h2non/semver.c	64
C.	DSPy Custom Signatures	68

List of Figures

3.1. OverHAuL Workflow	13
3.2. OverHAuL execution on dateparse	15
4.1. Benchmark Results	31
4.2. Iterations Heatmap	32

List of Listings

2.1. Fuzzing harness format	6
2.2. Example fuzzing harness	7
2.3. Compilation of harness	7
3.1. OverHAuL installation	22
3.2. Static analysis report	23
3.3. Codebase oracle samples	24
3.4. Generated compilation command	24
3.5. Sample dateparse harness	25
3.6. Sample dateparse crash input	26
3.7. Sample dateparse harness output	27
3.8. DSPy example	28

List of Tables

- 4.1. [The benchmark project corpus. Each project name links to its corresponding GitHub repository. Each is followed by a short description, its GitHub stars count and its Significant Lines of Code \(SLOC\), as of July 18th, 2025.](#) 30

1. Introduction

Modern society’s reliance on software systems continues to grow, particularly in mission-critical environments such as healthcare, aerospace, and industrial infrastructure. The reliability of these systems is crucial—failures or vulnerabilities can lead to severe financial losses and even endanger lives. A significant portion of this foundational software is still written in C, a language created by Dennis Ritchie in 1972 [1], [2]. Although C has been instrumental in the evolution of software, its lack of safeguards—especially around memory management—is notorious. Memory safety bugs remain a persistent vulnerability, and producing provably and verifiably safe code in C is exceptionally challenging—take for example the stringent guidelines required by organizations like NASA for safety-critical applications [3].

To address these challenges, programming languages with built-in memory safety features, such as Ada and Rust, have been introduced [4], [5]. Nevertheless, no language offers absolute immunity from such vulnerabilities. In addition, much of the global software infrastructure remains written in memory-unsafe languages, with C-based codebases unlikely to disappear in the near future. Ultimately, the potential for human error grows in tandem with increasing software complexity, meaning software is only as safe as its weakest link.

The advent of Large Language Models (LLMs) has profoundly influenced software development. Developers have begun to regularly use LLMs for code generation, refactoring, and documentation assistance. These models at large demonstrate remarkable programming capabilities. Still, they can often introduce subtle errors that may go unnoticed by even experienced developers. Many researchers argue that the use of such technologies inherently contributes to the generation of insecure code [6]–[8]. As LLM-generated code becomes more pervasive, so does the likelihood of unnoticed software errors escaping traditional human review.

Within this landscape, the need to detect vulnerabilities and ensure software quality is more urgent than ever. Fuzzing, a technique that generates and executes a vast array of test cases to identify potential bugs, has emerged as a vital approach for detecting memory safety violations. However, the necessity of manually-written harnesses—programs designed to exercise the Application Programming Interface (API) of the software under examination—poses a significant barrier to its broader adoption. As a result, the field of fuzzing automation through LLMs has gained considerable traction in recent years. Despite extensive advances in automating fuzzing, significant hurdles remain. Most current automatic-fuzzing systems require pre-existing fuzz harnesses [9] or depend on sample client code to exercise the target program [10]–[12]. Often, these tools still rely on developers for integration or final evaluation, leaving parts of the process manual and incomplete. Consequently, the application of LLMs to harness generation and end-to-end fuzzing remains a developing field.

This thesis aims to push the boundaries of fuzzing automation by leveraging the code synthesis and most importantly reasoning strengths of modern LLMs. We introduce OverHAuL, a system that accepts a bare and previously unfuzzed C project, utilizes LLM agents to author a new fuzzing harness from scratch and evaluates its efficacy in a closed iterative feedback loop. In this loop, said feedback is constantly utilized to improve the generated harness. This end-to-end approach is designed to minimize manual effort and accelerate vulnerability detection in C codebases.

1.1. Thesis Structure

This thesis begins by establishing the fundamental concepts required to contextualize its contributions (Chapter 2). Subsequently, we introduce the OverHAuL system, providing a comprehensive description of its architecture, the innovative techniques employed and their respective roles in advancing the state of automated harness generation (Chapter 3). In the evaluation chapter (Chapter 4), we assemble a benchmark suite comprised of ten open-source C projects and systematically evaluate the effectiveness of OverHAuL by measuring the number of successfully generated harnesses. Additionally, we present an extensive survey of recent research in automated fuzzing (Chapter 5), highlighting that most fuzzing systems either rely on pre-existing harnesses or employ client code, thereby shifting the responsibility for validation and integration onto the user. Finally, we discuss avenues for future enhancements to OverHAuL and conclude with a summary of our findings.

1.2. Summary of Contributions

This thesis presents the following key contributions:

1. The introduction of OverHAuL, a framework that enables fully automated end-to-end fuzzing harness generation using LLMs. It introduces novel techniques like an iterative feedback loop between LLM agents and the usage of a codebase oracle for code exploration.
2. Empirical validation through benchmarking experiments using ten real-world open source projects. We demonstrate that OverHAuL generates effective fuzzing harnesses with a success rate of **81.25%**.
3. Full open sourcing of all research artifacts, datasets, and code at <https://github.com/kchousos/OverHAuL> to encourage further research and ensure reproducibility.

This work aims to advance the use of LLMs in automated software testing, particularly for legacy codebases where building harnesses by hand is impractical or costly. By doing so, we strive to enhance software security and reliability in sectors where correctness is imperative.

2. Background

This chapter provides the foundational and necessary background for this thesis, by exploring the core concepts and technological advances central to modern fuzzing and Large Language Models (LLMs). It begins with an in-depth definition and overview of fuzz testing—an automated technique for uncovering software bugs and vulnerabilities through randomized input generation—highlighting its methodology, tools, and impact. What follows is a discussion on LLMs and their transformative influence on natural language processing, programming, and code generation. Challenges and opportunities in applying LLMs to tasks such as fuzzing harness generation are examined, leading to a discussion of Neurosymbolic AI, an emerging approach that combines neural and symbolic reasoning to address the limitations of current AI systems. This multifaceted background establishes the context necessary for understanding the research and innovations presented in subsequent chapters.

2.1. Fuzz Testing

Fuzzing is an automated software-testing technique in which a *Program Under Test* (PUT) is executed with (pseudo-)random inputs in the hope of exposing undefined behavior. When such behavior manifests as a crash, hang, or memory-safety violation, the corresponding input constitutes a *test-case* that reveals a bug and often a vulnerability [13]. In a certain sense, fuzzing is a form of adversarial, penetration-style testing carried out by the defender before the adversary has an opportunity to do so. Interest in the technique surged after the publication of three practitioner-oriented books in 2007–2008 [14]–[16].

Historically, the term was coined by Miller et al. in 1990, who used “fuzz” to describe a program that “generates a stream of random characters to be consumed by a target program” [17]. This informal usage captured the essence of what fuzzing aims to do: stress test software by bombarding it with unexpected inputs to reveal bugs. To formalize this concept, we adopt Manes et al.’s rigorous definitions [13]:

Definition 2.1 (Fuzzing). Fuzzing is the execution of a Program Under Test (PUT) using input(s) sampled from an input space (the *fuzz input space*) that protrudes the expected input space of the PUT [13].

This means fuzzing involves running the target program on inputs that go beyond those it is typically designed to handle, aiming to uncover hidden issues. An individual instance of such execution—or a bounded sequence thereof—is called a *fuzzing run*. When these runs are

conducted systematically and at scale with the specific goal of detecting violations of a security policy, the activity is known as *fuzz testing* (or simply *fuzzing*):

Definition 2.2 (Fuzz Testing). Fuzz testing is the use of fuzzing to test whether a PUT violates a security policy [13].

This distinction highlights that fuzz testing is fuzzing with an explicit focus on security properties and policy enforcement. Central to managing this process is the *fuzzer engine*, which orchestrates the execution of one or more fuzzing runs as part of a *fuzz campaign*. A fuzz campaign represents a concrete instance of fuzz testing tailored to a particular program and security policy:

Definition 2.3 (Fuzzer, Fuzzer Engine). A fuzzer is a program that performs fuzz testing on a PUT [13].

Definition 2.4 (Fuzz Campaign). A fuzz campaign is a specific execution of a fuzzer on a PUT with a specific security policy [13].

Throughout each execution within a campaign, a *bug oracle* plays a critical role in evaluating the program’s behavior to determine whether it violates the defined security policy:

Definition 2.5 (Bug Oracle). A bug oracle is a component (often inside the fuzzer) that determines whether a given execution of the PUT violates a specific security policy [13].

In practice, bug oracles often rely on runtime instrumentation techniques, such as monitoring for fatal POSIX signals (e.g., SIGSEGV) or using sanitizers like AddressSanitizer (ASan) [18]. Tools like LibFuzzer [19] commonly incorporate such instrumentation to reliably identify crashes or memory errors during fuzzing.

Most fuzz campaigns begin with a set of *seeds*—inputs that are well-formed and belong to the PUT’s expected input space—called a *seed corpus*. These seeds serve as starting points from which the fuzzer generates new test cases by applying transformations or mutations, thereby exploring a broader input space:

Definition 2.6 (Seed). An input given to the PUT that is mutated by the fuzzer to produce new test cases. During a fuzz campaign (Definition 2.4) all seeds are stored in a *seed pool* or *corpus* [13].

The process of selecting an effective initial corpus is crucial because it directly impacts how quickly and thoroughly the fuzzer can cover the target program’s code. This challenge—studied as the *seed-selection problem*—involves identifying seeds that enable rapid discovery of diverse execution paths and is non-trivial [20]. A well-chosen seed set often accelerates bug discovery and improves overall fuzzing efficiency.

2.1.1. Motivation

The purpose of fuzzing relies on the assumption that there are bugs within every program, which are waiting to be discovered. Therefore, a systematic approach should find them sooner or later.

— OWASP Foundation [21]

Fuzz testing provides several key advantages that contribute substantially to software quality and security. First, by uncovering vulnerabilities early in the development cycle, fuzzing reduces both the cost and risk associated with addressing security flaws after deployment. This proactive approach not only minimizes potential exposure but also streamlines the remediation process. Additionally, by subjecting software to the same randomized, adversarial inputs that malicious actors might use, fuzz testing puts defenders on equal footing with attackers, enhancing preparedness against emerging zero-day threats.

Beyond security, fuzzing plays a crucial role in improving the robustness and correctness of software systems. It is particularly effective at identifying logical errors and stability issues in complex, high-throughput APIs—such as decompressors and parsers—especially when these systems are expected to handle only well-formed inputs. Moreover, the integration of fuzz testing into continuous integration pipelines provides an effective guard against regressions. By systematically re-executing a corpus of previously discovered crashing inputs, developers can ensure that resolved bugs do not resurface in subsequent releases, thereby maintaining a consistent level of software reliability over time.

2.1.1.1. Success Stories

Heartbleed (CVE-2014-0160) [22], [23] arose from a buffer over-read¹ in the TLS implementation of the OpenSSL library [24], introduced on 1st of February 2012 and unnoticed until 1st of April 2014. Later analysis showed that a simple fuzz campaign exercising the TLS heartbeat extension would have revealed the defect almost immediately [25].

Likewise, the *Shellshock* (or *Bashdoor*) family of bugs in GNU Bash [26] enabled arbitrary command execution on many UNIX systems. While the initial flaw was fixed promptly, subsequent bug variants were discovered by Google’s Michał Zalewski using his own fuzzer—the now ubiquitous AFL fuzzer [27]—in late 2014 [28].

On the defensive tooling side, the security tool named *Mayhem* [29], [30]—developed by the company of the same name, formerly known as ForAllSecure—has since been adopted by the US Air Force, the Pentagon, Cloudflare, and numerous open-source communities. It has found and facilitated the remediation of thousands of previously unknown vulnerabilities, from errors in Cloudflare’s infrastructure to bugs in open-source projects like OpenWRT [31].

¹<https://xkcd.com/1354/> provides a concise illustration.

These cases underscore the central thesis of fuzz testing: exhaustive manual review is infeasible, but scalable stochastic exploration reliably surfaces the critical few defects that matter most.

2.1.2. Methodology

As previously discussed, fuzz testing of a PUT is typically conducted using a dedicated fuzzing engine (Definition 2.3). Among the most widely adopted fuzzers for C and C++ projects and libraries are AFL [27]—which has since evolved into AFL++ [32]—and LibFuzzer [19]. Within the OverHAuL framework, LibFuzzer is preferred due to its superior suitability for library fuzzing, whereas AFL++ predominantly targets executables and binary fuzzing.

2.1.2.1. LibFuzzer

LibFuzzer [19] is an in-process, coverage-guided evolutionary fuzzing engine primarily designed for testing libraries. It forms part of the LLVM ecosystem [33] and operates by linking directly with the library under evaluation. The fuzzer delivers mutated input data to the library through a designated fuzzing entry point, commonly referred to as the *fuzz target* or *harness*.

Definition 2.7 (Fuzz target). A function that accepts a byte array as input and exercises the application programming interface (API) under test using these inputs [19]. This construct is also known as a *fuzz driver*, *fuzzer entry point*, or *fuzzing harness*.

For the remainder of this thesis, the terms presented in Definition 2.7 will be used interchangeably.

To effectively validate an implementation or library, developers are required to author a fuzzing harness that invokes the target library’s API functions utilizing the fuzz-generated inputs. This harness serves as the principal interface for the fuzzer and is executed iteratively, each time with mutated input designed to maximize code coverage and uncover defects. To comply with LibFuzzer’s interface requirements, a harness must conform to the function signature shown in Listing 2.1. A more illustrative example of such a harness is provided in Listing 2.2.

Listing 2.1 This function receives the fuzzing input via a pointer to an array of bytes (*Data*) and its associated size (*Size*). Efficiency in fuzzing is achieved by invoking the API of interest within the body of this function, thereby allowing the fuzzer to explore a broad spectrum of behavior through systematic input mutation.

```
1 int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {  
2     DoSomethingInterestingWithData(Data, Size);  
3     return 0;  
4 }
```

Listing 2.2 This example demonstrates a minimal harness that triggers a controlled crash upon receiving HI! as input.

```
1 // test_fuzzer.cpp
2 #include <stdint.h>
3 #include <stddef.h>
4
5 extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
6     if (size > 0 && data[0] == 'H')
7         if (size > 1 && data[1] == 'I')
8             if (size > 2 && data[2] == '!')
9                 __builtin_trap();
10    return 0;
11 }
```

To compile and link such a harness with LibFuzzer, the Clang compiler—also part of the LLVM project [33]—must be used alongside appropriate compiler flags. For instance, compiling the harness in Listing 2.2 can be achieved as shown in Listing 2.3.

Listing 2.3 This example illustrates the compilation and execution workflow necessary for deploying a LibFuzzer-based fuzzing harness.

```
1 # Compile test_fuzzer.cc with AddressSanitizer and link against LibFuzzer.
2 clang++ -fsanitize=address,fuzzer test_fuzzer.cc
3 # Execute the fuzzer without any pre-existing seed corpus.
4 ./a.out
```

2.1.2.2. AFL and AFL++

American Fuzzy Lop (AFL) [27], developed by Michał Zalewski, is a seminal fuzzer targeting C and C++ applications. Its core methodology relies on instrumented binaries to provide edge coverage feedback, thereby guiding input mutation towards unexplored program paths. AFL supports several emulation backends including QEMU [34]—an open-source CPU emulator facilitating fuzzing on diverse architectures—and Unicorn [35], a lightweight multi-platform CPU emulator. While AFL established itself as a foundational tool within the fuzzing community, its successor AFL++ [32] incorporates numerous enhancements and additional features to improve fuzzing efficacy.

AFL operates by ingesting seed inputs from a specified directory (`seeds_dir`), applying mutations, and then executing the target binary to discover novel execution paths. Execution can be initiated using the following command-line syntax:

```
1 ./afl-fuzz -i seeds_dir -o output_dir -- /path/to/tested/program
```

AFL is capable of fuzzing both black-box and instrumented binaries, employing a fork-server mechanism to optimize performance. It additionally supports persistent mode execution as well as modes leveraging QEMU and Unicorn emulators, thereby providing extensive flexibility for different testing environments.

Although AFL is traditionally utilized for fuzzing standalone programs or binaries, it is also capable of fuzzing libraries and other software components. In such scenarios, rather than implementing the `LLVMFuzzerTestOneInput` style harness, AFL can use the standard `main()` function as the fuzzing entry point. Nonetheless, AFL also accommodates integration with `LLVMFuzzerTestOneInput`-based harnesses, underscoring its adaptability across varied fuzzing use cases.

2.1.3. Challenges in Adoption

Despite its potential for uncovering software vulnerabilities, fuzzing remains a relatively under-utilized testing technique compared to more established methodologies such as Test-Driven Development (TDD). This limited adoption can be attributed, in part, to the substantial initial investment required to design and implement appropriate test harnesses that enable effective fuzzing processes. Furthermore, the interpretation of fuzzing outcomes—particularly the identification, diagnostic analysis, and prioritization of program crashes—demands considerable resources and specialized expertise. These factors collectively pose significant barriers to the widespread integration of fuzzing within standard software development and testing practices. OverHAuL addresses this challenge by facilitating the seamless integration of fuzzing into developers’ workflows, minimizing initial barriers and reducing upfront costs to an almost negligible level.

2.2. Large Language Models

Natural Language Processing (NLP), a subfield of AI, has a rich and ongoing history that has evolved significantly since its beginning in the 1990s [36], [37]. Among the most notable—and recent—advancements in this domain are LLMs, which have transformed the landscape of NLP and AI in general.

At the core of many LLMs is the attention mechanism, which was introduced by Bahdanau et al. in 2014 [38]. This pivotal innovation enabled models to focus on relevant parts of the input sequence when making predictions, significantly improving language understanding and generation tasks. Building on this foundation, the Transformer architecture was proposed by Vaswani et al. in 2017 [39]. This architecture has become the backbone of most contemporary LLMs, as it efficiently processes sequences of data, capturing long-range dependencies without being hindered by sequential processing limitations.

One of the first major breakthroughs utilizing the Transformer architecture was BERT (Bidirectional Encoder Representations from Transformers), developed by Devlin et al. in 2019 [40]. BERT’s bi-directional understanding allowed it to capture the context of words from both directions, which improved the accuracy of various NLP tasks. Following this, the Generative Pre-trained Transformer (GPT) series, initiated by OpenAI with the original GPT model in 2018 [41], further pushed the boundaries. Subsequent iterations, including GPT-2 [42], GPT-3 [43], and the most current GPT-4 [44], have continued to enhance performance by scaling model size, data, and training techniques.

In addition to OpenAI’s contributions, other significant models have emerged, such as Claude, DeepSeek-R1 and the Llama series (1 through 3) [45]–[47]. The proliferation of LLMs has sparked an active discourse about their capabilities, applications, and implications in various fields.

2.2.1. State-of-the-art GPTs

User-facing LLMs are generally categorized between closed-source and open-source models. Closed-source LLMs like ChatGPT, Claude, and Gemini [45], [48], [49] represent commercially developed systems often optimized for specific tasks without public access to their underlying weights. In contrast, open-source models², including the Llama series [47] and Deepseek [46], provide researchers and practitioners with access to model weights, allowing for greater transparency and adaptability.

2.2.2. Prompting

Interaction with LLMs typically occurs through chat-like interfaces where the user gives queries and tasks for the LLM to answer and complete, a process commonly referred to as *prompting*. A critical aspect of effective engagement with LLMs is the usage of different prompting strategies, which can significantly influence the quality and relevance of the generated outputs. Various approaches to prompting have been developed and studied, including zero-shot and few-shot prompting. In zero-shot prompting, the model is expected to perform the given task without any provided examples, while in few-shot prompting, the user offers a limited number of examples to guide the model’s responses [43].

To enhance performance on more complex tasks, several advanced prompting techniques have emerged. One notable strategy is the *Chain of Thought* approach (COT) [50], which entails presenting the model with sample thought processes for solving a given task. This method encourages the model to generate more coherent and logical reasoning by mimicking human-like cognitive pathways. A more refined but complex variant of this approach is the *Tree*

²The term “open-source” models is somewhat misleading, since these are better termed as *open-weights* models. While their weights are publicly available, their training data and underlying code are often proprietary. This terminology reflects community usage but fails to capture the limitations of transparency and accessibility inherent in these models.

of *Thoughts* technique [51], which enables the LLM to explore multiple lines of reasoning concurrently, thereby facilitating the selection of the most promising train of thought for further exploration.

In addition to these cognitive strategies, Retrieval-Augmented Generation (RAG) [52] is another innovative technique that enhances the model’s capacity to provide accurate information by incorporating external knowledge not present in its training dataset. RAG operates by integrating the LLM with an external storage system—often a vector store containing relevant documents—that the model can query in real-time. This allows the LLM to pull up pertinent and/or proprietary information in response to user queries, resulting in more comprehensive and accurate answers.

Moreover, the ReAct framework [53], which stands for Reasoning and Acting, empowers LLMs by granting access to external tools. This capability allows LLM instances to function as intelligent agents that can interact meaningfully with their environment through user-defined functions. For instance, a ReAct tool could be a function that returns a weather forecast based on the user’s current location. In this scenario, the LLM can provide accurate and truthful predictions, thereby mitigating risks associated with hallucinated responses.

2.2.3. LLMs for Coding

The impact of LLMs in software development in recent years is apparent, with hundreds of LLM-assistance extensions and Integrated Development Environments (IDEs) being published. Notable instances include tools like GitHub Copilot and IDEs such as Cursor [54], [55], which leverage LLM capabilities to provide developers with coding suggestions, auto-completions, and even real-time debugging assistance. Such innovations have introduced a layer of interaction that enhances productivity and fosters a more intuitive coding experience. Additionally, more and more LLMs are now specifically trained for usage in code-generation tasks [56]–[58].

One exemplary product of this innovation is *vibecoding* and the no-code movement, which describe the development of software by only prompting and tasking an LLM, i.e. without any actual programming required by the user. This constitutes a showcase of how LLMs can be used to elevate the coding experience by supporting developers as they navigate complex programming tasks [59]. By analyzing the context of the code being written, these sophisticated models can provide contextualized insights and relevant snippets, effectively streamlining the development process. Developers can benefit from reduced cognitive load, as they receive suggestions that not only cater to immediate coding needs but also promote adherence to best practices and coding standards.

Despite these advancements, it is crucial to recognize the inherent limitations of LLMs when applied to software development. While they can help in many aspects of coding, they are not immune to generating erroneous outputs—a phenomenon often referred to as “hallucination” [60]. Hallucinations occur when LLMs produce information that is unfounded or inaccurate, which can stem from several factors, including the limitations of their training data and the constrained context window within which they operate. As LLMs generate code suggestions

based on the patterns learned from vast datasets, they may inadvertently propose solutions that do not align with the specific requirements of a task or that utilize outdated programming paradigms.

Moreover, the challenge of limited context windows can lead to suboptimal suggestions [61]. LLMs generally process a fixed amount of text when generating responses, which can impact their ability to fully grasp the nuances of complex coding scenarios. This may result in outputs that lack the necessary depth and specificity required for successful implementation. As a consequence, developers must exercise caution and critically evaluate the suggestions offered by these models, as reliance on them without due diligence could lead to the introduction of bugs or other issues in the code.

2.2.4. LLMs for Fuzzing

In the domain of fuzzing, recent research has explored the application of LLMs primarily along two axes: employing LLMs to generate seeds and inputs for the program under test [62]–[65] and leveraging them to generate the fuzz driver itself (Chapter 5). This thesis focuses on the latter, recognizing that while using LLMs for seed generation offers certain advantages, the challenge of automating harness generation represents a deeper and more meaningful frontier. Significant limitations such as restricted context windows and the propensity for LLMs to hallucinate remain central concerns in this area [60], [61].

The process of constructing a fuzzing harness is inherently complex, demanding a profound understanding of the target library and the nuanced interactions among its components. Achieving this level of comprehension is often beyond the reach of LLMs when deployed in isolation. Empirical evidence by Jiang et al. [66] demonstrates that zero-shot harness generation with LLMs is both ineffective and prone to significant errors. Specifically, LLMs tend to rely heavily on patterns encountered during training, which leads to the erroneous invocation of APIs, particularly when their context window is pushed to its limits. This over-reliance on training data exacerbates the risk of hallucination, compounding the challenge of generating correct and robust fuzz drivers.

Compounding this issue is the inherent risk introduced by error-prone code synthesized by LLMs. In the context of fuzzing, a fundamental requirement is the clear attribution of observed failures: developers must be confident that detected crashes stem from vulnerabilities in the tested software rather than flaws or bugs inadvertently introduced by the harness. This necessity imposes an additional verification burden, increasing developers’ cognitive load and diverting attention from the primary goal of meaningful software evaluation and improvement.

Enhancing the reliability of LLM-generated harnesses thus necessitates systematic and programmatic evaluation and validation of generated artifacts [67]. Such validation involves implementing structured techniques to rigorously assess both the accuracy and robustness of the code, confirming that it interacts correctly with the relevant software components and behaves as intended. This approach aligns with the emerging framework of Neurosymbolic AI (Section 2.3), which integrates the statistical learning capabilities of neural networks with

the rigor and precision of symbolic reasoning. By leveraging the strengths of both paradigms, neuroscience-inspired symbolic methods [68] may offer pathways toward more reliable and effective LLM-generated fuzzing harnesses, facilitating a smoother integration of automated testing practices into established software development pipelines [69], [70].

2.3. Neurosymbolic AI

Neurosymbolic AI represents a groundbreaking fusion of neural network methodologies with symbolic execution techniques and tools, providing a multi-faceted approach to overcoming the inherent limitations of traditional AI paradigms [71], [72]. This innovative synthesis seeks to combine the strengths of both neural networks, which excel in pattern recognition and data-driven learning, and symbolic systems, which offer structured reasoning and interpretability. By integrating these two approaches, neurosymbolic AI aims to create cognitive models that are not only more accurate but also more robust in problem-solving contexts.

At its core, Neurosymbolic AI facilitates the development of AI systems that are capable of understanding and interpreting feedback in real-world scenarios [73]. This characteristic is particularly significant in the current landscape of artificial intelligence, where LLMs are predominant. In this context, Neurosymbolic AI is increasingly viewed as a critical solution to pressing issues related to explainability, attribution, and reliability in AI systems [67], [74]. These challenges are essential for ensuring that AI systems can be trusted and effectively utilized in various applications, from business to healthcare.

The burgeoning field of neurosymbolic AI is still in its nascent stages, with ongoing research and development actively exploring its potential to enhance attribution methodologies within large language models. By addressing these critical challenges, Neurosymbolic AI can significantly contribute to the broader landscape of trustworthy AI systems, allowing for more transparent and accountable decision-making processes [67], [71], [74].

Moreover, the application of neurosymbolic AI within the domain of fuzzing is gaining traction, paving the way for innovative explorations. This integration of LLMs with symbolic systems opens up new avenues for research. Currently, there are only a limited number of tools that support such hybrid approaches (Chapter 5). Among these, OverHAuL constitutes a Neuro[Symbolic] tool, as classified by Henry Kautz’s taxonomy [75], [76]. This means that the neural model—specifically the LLM—can leverage symbolic reasoning tools—in this case a source code explorer (Section 3.6)—to augment its reasoning capabilities. This symbiotic relationship enhances the overall efficacy and versatility of LLMs for fuzzing harnesses generation, demonstrating the profound potential held by the fusion of neural and symbolic methodologies.

3. OverHAuL’s Design

In this thesis we present **OverHAuL** (**H**arness **A**utomation with **L**LMs), a neurosymbolic AI tool that automatically generates fuzzing harnesses for C libraries through LLM agents. In its core, OverHAuL is comprised by a triplet of LLM ReAct agents [53]—each with its own responsibility and scope—and a codebase oracle reserving the given project’s analyzed source code. An overview of OverHAuL’s process is presented in Figure 3.1, detailed in Section 3.2. The objective of OverHAuL is to streamline the process of fuzz testing for unfuzzed C libraries. Given a link to a git repository [77] of a C library, OverHAuL automatically generates a new fuzzing harness specifically designed for the project. In addition to the harness, it produces a compilation script to facilitate building the harness, generates a representative input that can trigger crashes, and logs the output from the executed harness.

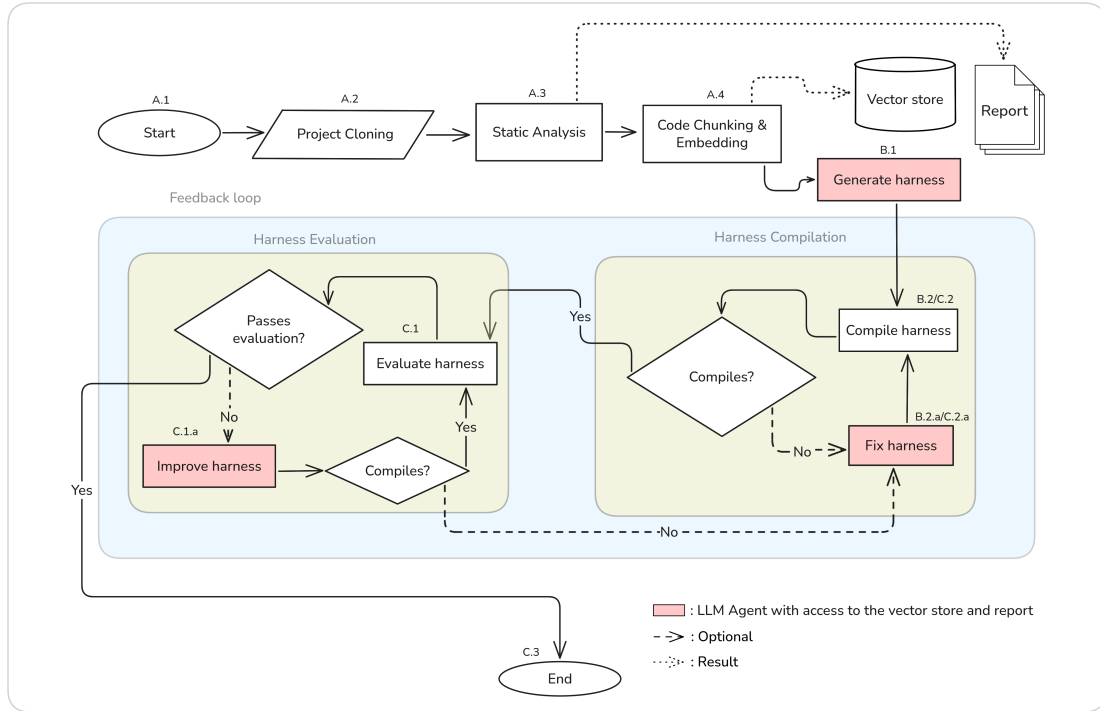


Figure 3.1: Overview of OverHAuL’s automatic harnessing process.

OverHAuL utilizes autonomous ReAct agents which inspect and analyze the project’s source code. The latter is stored and interacted with as a set of text embeddings [78], kept in a vector store. Both approaches are, to the best of our knowledge, novel in the field of automatic fuzzing

harnesses generation. OverHAuL also implements an evaluation component that assesses in real-time all generated harnesses, making the results tenable, reproducible and well-founded. Ideally, this methodology provides a comprehensive and systematic framework for identifying previously unknown software vulnerabilities in projects that have not yet been fuzz tested.

As detailed in Section 5.4, OverHAuL does not expect and depend on the existence of client code or unit tests [10]–[12] *nor* does it require any preexisting fuzzing harnesses [9] or any documentation present [79]. Also importantly, OverHAuL is decoupled from other fuzzing projects, thus lowering the barrier to entry for new projects [9], [80]. Lastly, the user isn’t mandated to manually specify the function which the harness-to-be-generated must fuzz. Instead, OverHAuL’s agents examine and assess the provided codebase, choosing after evaluation the most optimal target function.

Finally, OverHAuL excels in its user-friendliness, as it constitutes a simple and easily-installable Python package with minimal external dependencies—only real dependency being Clang, a prevalent compiler available across all primary operating systems. This contrasts most other comparable systems, which are typically characterized by their limited documentation, lack of extensive testing, and a focus primarily on experimental functionality.¹

3.1. Installation and Usage

The source code of OverHAuL is available in <https://github.com/kchousos/OverHAuL>. OverHAuL can be installed by cloning the git repository locally, creating and enabling a Python3.10 virtual environment [82] (optional, but recommended) and installing it inside the environment using Python’s PIP package installer [83], like in Listing 3.1.

To use OverHAuL, you need to provide a secret key for using OpenAI’s API service. This key can be either stored in a `.env` file in the root directory or exported in the shell environment:

```
1 $ echo "OPENAI_API_KEY=<API-key-here>" >> .env
2 # OR
3 $ export OPENAI_API_KEY=<API-key-here>
```

Once these preliminary steps are completed, OverHAuL can be executed. The primary argument required by OverHAuL is the repository link of the library that is to be fuzzed. Additionally, users have the option to specify certain command-line flags, which allow them to control the checked-out commit of the cloned project, select the OpenAI LLM model from a predefined list, define specific file patterns for OverHAuL to search for, and determine the directory in which the project will be cloned. For a concrete example, we will use OverHAuL to create a

¹For instance, both fuzz-introspector and OSS-Fuzz-Gen are integrated by design to the OSS-Fuzz platform [9], [80], [81]. When utilized outside this environment, they require users to operate directly from the project’s root directory and interact with the tools primarily through unrefined Python scripts, thereby limiting their accessibility and ease of use.

new fuzzing harness for [dvhar's dateparsing C library](#) and specify the LLM model to OpenAI's gpt-4.1 model. The resulting command and its output is presented in Figure 3.2.

```
> overhauL https://github.com/dvhar/dateparse -m gpt-4.1
2025-07-17 00:55:18.887 | INFO | overhauL.cli:parse_arguments:184 - Cloning project's repo in the output/dateparse directory...
2025-07-17 00:55:18.976 | INFO | overhauL.core:harnesser:_init_:167 - Initializing LLM...
2025-07-17 00:55:18.991 | INFO | overhauL.cli:main:243 - Iteration 1 of harnessing...
2025-07-17 00:55:18.991 | INFO | overhauL.core:harnesser:harness:287 - Calling LLM to generate a harness...
2025-07-17 00:55:22.837 | DEBUG | overhauL.rag:ragger:rag_tool:88 - Agent using RAG: "Show the functions in dateparse.c that contain a strcpy call, especially near lines 485, 888, and 1242."
2025-07-17 00:55:25.887 | DEBUG | overhauL.rag:ragger:rag_tool:88 - Agent using RAG: "Provide the full implementation (signature and body) of the dateparse function in dateparse.c."
2025-07-17 00:55:30.287 | DEBUG | overhauL.rag:ragger:rag_tool:88 - Agent using RAG: "Show the full signature and body of the function int dateparse(const char* datestr, date_t* t, int *offset, int stringlen) in dateparse.c."
2025-07-17 00:55:34.176 | DEBUG | overhauL.rag:ragger:rag_tool:88 - Agent using RAG: "Show the definition of struct parser and the typedef of date_t as used in dateparse.c and dateparse.h."
2025-07-17 00:55:38.581 | DEBUG | overhauL.rag:ragger:rag_tool:88 - Agent using RAG: "Show the definition of struct parser and the typedef of date_t from dateparse.h."
2025-07-17 00:56:01.897 | INFO | overhauL.io:file_manager:write_harness:68 - Writing harness to project...
2025-07-17 00:56:01.898 | INFO | overhauL.io:file_manager:write_harness:92 - Harness written to output/dateparse/harnesses/harness.c
2025-07-17 00:56:01.898 | INFO | overhauL.core:builder:build_harness:63 - Building harness...
2025-07-17 00:56:01.899 | INFO | overhauL.core:builder:build_harness:148 - Starting compilation of harness: harnesses/harness.c
2025-07-17 00:56:02.345 | INFO | overhauL.core:builder:build_harness:149 - Harness compiled successfully
2025-07-17 00:56:02.345 | INFO | overhauL.core:evaluator:evaluate_harness:81 - Evaluating harness...
2025-07-17 00:56:02.345 | INFO | overhauL.core:evaluator:evaluate_harness:98 - Starting execution of harness...
2025-07-17 00:56:02.417 | INFO | overhauL.core:evaluator:evaluate_harness:119 - Harness execution completed in 0.07 seconds.
2025-07-17 00:56:02.419 | INFO | overhauL.core:evaluator:evaluate_harness:181 - New testcases created (1): (('crash-dfaa34d8e98889cd82dcd688cf96fd84552a2b4', 1752782962.4113252))
2025-07-17 00:56:02.419 | SUCCESS | overhauL.cli:main:282 - All done!
```

Figure 3.2.: A successful execution of OverHAuL, harnessing the “dateparse” library using OpenAI’s gpt-4.1 model. Debug statements are printed to showcase the queries of the LLM agents to the codebase oracle (Section 3.3.3).

In this example, the dateparse repository is cloned into the `./output/dateparse` directory, which is relative to the root directory of OverHAuL. Following a successful execution, the project’s directory will contain a new folder named `harnesses`, which will house all the generated harnesses formatted as `harness_n.c`—where n ranges from 1 to $N - 1$, with N representing the total number of harnesses produced. The most recent and verifiably correct harness will be designated simply as `harness.c`. Additionally, the dateparse folder will include an executable script named `overhauL.sh`, which contains the compilation command necessary for the harness. A log file titled `harness.out` will also be present, documenting the output from the latest harness execution. Lastly and most importantly, there will be at least one non-empty crash file included, serving as a witness to the harness’s correctness. In the following sections, the intermediary steps between invocation and completion are dissected and analyzed. The dateparse project is used as a running example.

3.2. Architecture

OverHAuL can be compartmentalized in three stages: First, the project analysis stage (Section 3.2.1), the harness creation stage (Section 3.2.2) and the harness evaluation stage (Section 3.2.3).

3.2.1. Project Analysis

In the project analysis stage (steps A.1–A.4), dateparse is ran through a static analysis tool named Flawfinder [84] and is sliced into function-level chunks, which are stored in a vector store. The results of this stage are a *static analysis report* and a *codebase oracle*, i.e. a vector store containing embeddings of function-level code chunks. Both resources are later available to the LLM agents. Flawfinder is executed with the dateparse directory as input and is responsible for the static analysis report. This report is considered a meaningful resource, since it provides the LLM agent responsible with the harness creation with some starting points to explore,

regarding the occurrences of potentially vulnerable functions and/or unsafe code practices. Part of dateparse’s static analysis report is shown in Listing 3.2.

The codebase oracle is created in the following manner: The source code is first chunked in function-level pieces by traversing the code’s Abstract Syntax Tree (AST) through Clang. Each chunk is represented by an object with the function’s signature, the corresponding filepath and the function’s body (see Listing 3.3). Afterwards, each function body is turned into a vector embedding through an embedding model. Each embedding is stored in the vector store. This structure is created and used for easier and more semantically meaningful code retrieval, and to also combat context window limitations present in LLMs.

3.2.2. Harness Creation

Second is the harness creation stage (steps B.1–B.2). In this part, a “generator” ReAct LLM agent is tasked with creating a fuzzing harness for the project. The agent has access to a querying tool that acts as an interface between it and the codebase oracle. When the agent makes queries like “functions containing strcpy()”, the querying tool turns the question into an embedding and through similarity search returns the top $k = 5$ most similar results—in this case, functions of the project. With this approach, the agent is able to explore the codebase semantically and pinpoint potentially vulnerable usage patterns easily.

The harness generated by the agent is then compiled using Clang and linked with the AddressSanitizer, LeakSanitizer, and UndefinedBehaviorSanitizer. The compilation command used is generated programmatically, according to the rules described in Section 3.5. If the compilation fails for any reason, e.g. a missing header include, then the generated faulty harness and its compilation output are passed to a new “fixer” agent tasked with repairing any errors in the harness (step B.2.a). This results in a newly generated harness, presumably free from the previously shown flaws. This process is iterated until a compilable harness has been obtained. After success, a script is also exported in the project directory, containing the generated compilation command. Dateparse’s compilation command is shown in Listing 3.4.

3.2.3. Harness Evaluation

Third comes the evaluation stage (steps C.1–C.3). During this step, the compiled harness is executed and its results evaluated. Namely, a generated harness passes the evaluation phase if and only if:

1. The harness has no memory leaks during its execution

This is inferred by the existence of `leak-<hash>` files.

2. A new testcase was created *or* the harness executed for at least `MIN_EXECUTION_TIME` (i.e. did not crash on its own)

When a crash happens, and thus a testcase is created, it results in a `crash-<hash>` file.

3. The created testcase is not empty

This is examined through xxd’s output given the crash-file.

Similarly to the second stage’s compilation phase (steps B.2–B.2.a), if a harness does not pass the evaluation for whatever reason it is sent to an “improver” agent. This agent is instructed to refine it based on its code and cause of failing the evaluation. This process is also iterative. If any of the improved harness versions fail to compile, the aforementioned “fixer” agent is utilized again (steps C.2–C.2.a). All produced crash files and the harness execution output are saved in the project’s directory. An evaluation-passing harness generated for the dateparse project is presented in Listing 3.5, along with the associated crash input and execution output displayed in Listing 3.6 and Listing 3.7, respectively.

3.3. OverHAuL Techniques

The fundamental techniques that distinguish OverHAuL in its approach and enhance its effectiveness in achieving its objectives are: The implementation of an iterative feedback loop between the LLM agents, the distribution of responsibility across a triplet of distinct agents and the employment of a “codebase oracle” for interacting with the given project’s source code.

3.3.1. Feedback Loop

The initial generated harness produced by OverHAuL is unlikely to be successful from the get-go. The iterative feedback loop implemented facilitates its enhancement, enabling the harness to be tested under real-world conditions and subsequently refined based on the results of these tests. This approach mirrors the typical workflow employed by developers in the process of creating and optimizing fuzz targets.

In this iterative framework, the development process continues until either an acceptable and functional harness is realized or the defined *iteration budget* is exhausted. The iteration budget $N = 10$ is initialized at the onset of OverHAuL’s execution and is shared between both the compilation and evaluation phases of the harness development process. This means that the iteration budget is decremented each time a dashed arrow in the flowchart illustrated in Figure 3.1 is followed. Such an approach allows for targeted improvements while maintaining oversight of resource allocation throughout the harness development cycle.

3.3.2. React Agents Triplet

An integral design decision in our framework is the implementation of each agent as a distinct LLM instance, although all utilizing the same underlying model. This approach yields several advantages, particularly in the context of maintaining separate and independent contexts for each agent throughout each OverHAuL run.

By assigning individual contexts to the agents, we enable a broader exploration of possibilities during each run. For instance, the “improver” agent can investigate alternative pathways or strategies that the “generator” agent may have potentially overlooked or internally deemed inadequate inaccurately. This separation not only fosters a more diverse range of solutions but also enhances the overall robustness of the system by allowing for iterative refinement based on each agent’s unique insights.

Furthermore, this design choice effectively addresses the limitations imposed by context window sizes. By distributing the “cognitive” load across multiple agents, we can manage and mitigate the risks associated with exceeding these constraints. As a result, this architecture promotes efficient utilization of available resources while maximizing the potential for innovative outcomes in multi-agent interactions. This layered approach ultimately contributes to a more dynamic and exploratory research environment, facilitating a comprehensive examination of the problem space.

3.3.3. Codebase Oracle

The third central technique employed is the creation and utilization of a codebase oracle, which is effectively realized through a vector store. This oracle is designed to contain the various functions within the project, enabling it to return the most semantically similar functions upon querying it. Such an approach serves to address the inherent challenges associated with code exploration difficulties faced by LLM agents, particularly in relation to their limited context window.

By structuring the codebase into chunks at the level of individual functions, LLM agents can engage with the code more effectively by focusing on its functional components. This methodology not only allows for a more nuanced understanding of the codebase but also ensures that the responses generated do not consume an excessive portion of the limited context window available to the agents. In contrast, if the codebase were organized and queried at the file level, the chunks of information would inevitably become larger, leading to an increase in noise and a dilution of meaningful content in each chunk [85]. Given the constant size of the embeddings used in processing, each progressively larger chunk would be less semantically significant, ultimately compromising the quality of the retrieval process.

Defining the function as the primary unit of analysis represents the most proportionate balance between the size of the code segments and their semantic significance. It serves as the ideal “zoom-in” level for the exploration of code, allowing for greater clarity and precision in understanding the functionality of individual code segments. This same principle is widely recognized in the training of code-specific LLMs, where a function-level approach has been shown to enhance performance and comprehension [86]. By adopting this methodology, we aim to foster a more robust interaction between LLM agents and the underlying codebase, ultimately facilitating a more effective and efficient exploration process.

3.4. High-Level Algorithm

A pseudocode version of OverHAuL’s main function is shown in Algorithm 3.1, illustrating the workflow depicted in Figure 3.1 and incorporating the methods explained in Sections 3.2 and 3.3. Notably, within this algorithm, the `HarnessAgents()` function acts as an interface that connects the “generator”, “fixer”, and “improver” LLM agents. The specific agent utilized during each invocation of `HarnessAgents()` depends on the function’s arguments. As a result, the *harness* variable encapsulates all generated, fixed, or improved harnesses. Since both the “fixer” and “generator” agents are accessed through the `HarnessAgents()` function, the related continue statements correspond to the next iterations of fixing or improving a harness. This design choice streamlines the overall algorithm, making it more abstract and easier to comprehend.

Algorithm 3.1 OverHAuL

Require: *repository*

Ensure: *harness, compilation_script, crash_input, execution_log*

```
1: path ← REPOCLONE(repository)
2: report ← STATICANALYSIS(path)
3: vector_store ← CREATEORACLE(path)
4: acceptable ← False
5: compiled ← False
6: error ← None
7: violation ← None
8: out put ← None
9: for i = 1 to MAX_ITERATIONS do
10:   harness ← HARNESSAGENTS(path, report, vector_store, error, violation, out put)
11:   error, compiled ← BUILDHARNESS(path, harness)
12:   if  $\neg$ compiled then
13:     continue ▷ Fix harness
14:   end if
15:   out put, accepted ← EVALUATEHARNESS(path, harness)
16:   if  $\neg$ accepted then
17:     continue ▷ Improve harness
18:   else
19:     acceptable ← True
20:     break
21:   end if
22: end for
23: return compiled  $\wedge$  acceptable
```

3.5. Scope

Currently, OverHAuL is designed to generate new harnesses specifically for medium-sized C libraries. Given the inherent complexity of dealing with C++ projects, this is not a feature yet supported within the system.

The compilation command utilized by OverHAuL is created programmatically. It incorporates the root directory along with all subdirectories that conform to a predefined set of common naming conventions. Additionally, the compilation process uses all C source files identified within these directories. Crucially, it is important that no `main()` function is present in any of the files to ensure successful compilation. For this reason any files or directories that include “test”, “main”, “example”, “demo”, or “benchmark” in their paths are systematically excluded from the compilation process. This exclusion also decreases the “noise” in the oracle, as these files do not constitute part of the core library and would therefore not contain any functions meaningful to the LLM agents.

Lastly, No support for build systems such as Make or CMake [87], [88] is yet implemented. Such functionality would exponentially increase the complexity of the build step and is beyond the scope of this thesis.

3.6. Implementation

In creating the codebase oracle, we employ the “libclang” Python package [89] to slice functions based on the AST capability provided by Clang. As detailed in Section 3.3.3, the intermediate output consists of a list of Python dictionaries, with each dictionary storing a function’s body, signature, and corresponding file path. Each chunk of function code is then converted into an embedding using OpenAI’s “text-embedding-3-small” model [90] and stored in a FAISS vector store index [91]. This index is mapped to a metadata structure that contains the aforementioned function data—specifically the actual function body, signature, and file path. When a search is conducted on the index, the results returned are the embeddings. The responses that the LLM agent receives are derived from the corresponding metadata entries of each embedding.

All LLM agents and components are developed using the DSPy library, a declarative Python framework for LLM programming created by Stanford’s NLP research team [92]. DSPy offers built-in modules and abstractions that facilitate the composition of LLMs and prompting techniques, such as Chain of Thought and ReAct (Listing 3.8). Each agent within OverHAuL is an instance of DSPy’s ReAct module [93], accompanied by a custom Signature [94]—displayed in Appendix C. DSPy was selected over other contemporary LLM libraries, such as LangChain and Llamaindex [95], [96], because of its user-friendliness, logical abstractions, and efficient development process—qualities that are often lacking in these alternative libraries [97]–[99].

Repository cloning is executed using the `--depth 1` flag to minimize disk storage usage and reduce the size of artifacts.

The current implementation of OverHAuL sits at 1,254 source lines of Python code.

3.6.1. Development Tools

The development of OverHAuL incorporates a variety of tools aimed at enhancing functionality and efficiency. Notably, “uv” is a Python package and project manager written in Rust that serves as a replacement for Poetry. Additionally, “Ruff,” a code linter and formatter also developed in Rust, contributes to code quality by enforcing consistent formatting standards. The project also employs “MyPy,” the widely-used static type checker for Python, to ensure type correctness. Testing is facilitated through “PyTest,” a robust Python testing framework. Lastly, “pdoc” is utilized as a Static Site Generator (SSG) to automate the creation of API documentation² [100]–[104].

3.6.2. Reproducibility

OverHAuL’s source code is available at <https://github.com/kchousos/OverHAuL>. Each benchmark run was conducted within the framework of a GitHub Actions workflow, resulting in a detailed summary accompanied by an artifact containing all cloned repositories. These artifacts are the compressed result directories described in Section 4.1.1 and provide the essential components necessary for the reproducibility each project’s results, as described in Section 3.1. All benchmark runs can be conveniently accessed at <https://github.com/kchousos/OverHAuL/actions/workflows/benchmarks.yml>.

²Available at <https://kchousos.github.io/OverHAuL/>.

Listing 3.1 OverHAuL's straightforward installation process.

```
1 $ git clone https://github.com/kchousos/overhaul; cd overhaul
2 ...
3 $ python3.10 -m venv .venv
4 $ source ./venv/bin/activate
5 $ pip install .
6 ...
7 $ overhaul --help
8 usage: overhaul [-h] [-c COMMIT] [-m MODEL] [-f FILES [FILES ...]]
9 [-o OUTPUT_DIR] repo
10
11 Generate fuzzing harnesses for C/C++ projects
12
13 positional arguments:
14   repo                  Link of a project's git repo, for which to generate
15                        a harness.
16
17 options:
18   -h, --help            show this help message and exit
19   -c COMMIT, --commit COMMIT
20                        A specific commit of the project to check out
21   -m MODEL, --model MODEL
22                        LLM model to be used. Available: o3-mini, o3, gpt-4o,
23                        gpt-4o-mini, gpt-4.1, gpt-4.1-mini, gpt-3.5-turbo, gpt-4
24   -f FILES [FILES ...], --files FILES [FILES ...]
25                        File patterns to include in analysis (e.g. *.c *.h)
26   -o OUTPUT_DIR, --output-dir OUTPUT_DIR
27                        Directory to clone the project into. Defaults to "output"
28 $
```

Listing 3.2 Static analysis report (Flawfinder output) of dateparse.

```
1 Flawfinder version 2.0.19, (C) 2001-2019 David A. Wheeler.
2 Number of rules (primarily dangerous function names) in C/C++ ruleset: 222
3 Examining ./dateparse.c
4 Examining ./dateparse.h
5 Examining ./test.c
6
7 FINAL RESULTS:
8
9 ./dateparse.c:405: [4] (buffer) strcpy:
10   Does not check for buffer overflows when copying to destination [MS-banned]
11   (CWE-120). Consider using snprintf, strcpy_s, or strncpy (warning: strncpy
12   easily misused).
13
14 .....
15
16 ./dateparse.c:2192: [1] (buffer) strlen:
17   Does not handle strings that are not \0-terminated; if given one it may
18   perform an over-read (it could cause a crash if unprotected) (CWE-126).
19
20 ANALYSIS SUMMARY:
21
22 Hits = 64
23 Lines analyzed = 2719 in approximately 0.04 seconds (61234 lines/second)
24 Physical Source Lines of Code (SLOC) = 1966
25 Hits@level = [0] 15 [1] 28 [2] 31 [3] 0 [4] 5 [5] 0
26 Hits@level+ = [0+] 79 [1+] 64 [2+] 36 [3+] 5 [4+] 5 [5+] 0
27 Hits/KSLOC@level+ = [0+] 40.1831 [1+] 32.5534 [2+] 18.3113 [3+] 2.54323
28   [4+] 2.54323 [5+] 0
29 Dot directories skipped = 1 (--followdotdir overrides)
30 Minimum risk level = 1
31
32 Not every hit is necessarily a security vulnerability.
33 You can inhibit a report by adding a comment in this form:
34 // flawfinder: ignore
35 Make *sure* it's a false positive!
36 You can use the option --neverignore to show these.
37
38 There may be other security vulnerabilities; review your code!
39 See 'Secure Programming HOWTO'
40 (https://dwheeler.com/secure-programs) for more information.
```

Listing 3.3 Sample chunks contained in dateparse’s codebase oracle.

```
1 File: dateparse/dateparse.c
2 Signature: void (struct parser *, int, int)
3 Code:
4 static void setOffset(struct parser* p, int i, int len){
5     strncpy(p->offsetbuf, p->datestr+i, len);
6     p->offsetbuf[len] = 0;
7 }
8
9
10 File: dateparse/dateparse.c
11 Signature: void (struct parser *, char *)
12 Code:
13 static void setFullMonth(struct parser* p, char* month){
14     strcpy(p->mobuf, month);
15 }
16
17
18 File: dateparse/dateparse.c
19 Signature: int (const char *, long long *, int *, int)
20 Code:
21 int dateparse(const char* datestr, date_t* t, int *offset, int stringlen){
22     struct parser p;
23     *t = 0;
24     if (!stringlen)
25         stringlen = strlen(datestr);
26     if (parseTime(datestr, &p, stringlen))
27         return -1;
28     return parse(&p, t, offset);
29 }
```

Listing 3.4 OverHAuL’s generated compilation command for dateparse.

```
1 # cat ./overhaul.sh
2 clang -g -fsanitize=fuzzer,address,undefined harnesses/harness.c -I .
3 dateparse.c -o harness
```

Listing 3.5 A crash-finding harness for dateparse, generated through OverHAuL (some comments were removed).

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdint.h>
4  #include "dateparse.h"
5
6  // ...
7  struct parser {
8      char mobuf[16];
9  };
10
11 // ...
12 static void setFullMonth(struct parser* p, char* month){
13     strcpy(p->mobuf, month);
14 }
15
16 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
17     // Allocate a parser instance on the heap.
18     struct parser *p = (struct parser*)malloc(sizeof(struct parser));
19     if (!p) {
20         return 0;
21     }
22     // Initialize parser with zeros.
23     memset(p, 0, sizeof(struct parser));
24     // Prepare month string input: ensure null-terminated string for strcpy.
25     char *month = (char*)malloc(size + 1);
26     if (!month) {
27         free(p);
28         return 0;
29     }
30     memcpy(month, data, size);
31     month[size] = '\0'; // Null terminate to avoid overread in strcpy.
32     // Call the vulnerable function with fuzzed month string.
33     setFullMonth(p, month);
34     // Cleanup
35     free(month);
36     free(p);
37     return 0;
38 }
```

Listing 3.6 An input string that crashes the harness in Listing [3.5](#). What is shown is its xxd output.

```
00000000: 315e 5e5e 5e5e 5e5e 5e5e 5e5e 5e5e 5e5e  1^^^^^^^^^^^^^^^^  
00000010: 0a                                     .
```

Listing 3.7 The output of the harness in Listing 3.5 when executed with Listing 3.6 as input.

```
1  INFO: Running with entropic power schedule (0xFF, 100).
2  INFO: Seed: 2365219758
3  INFO: Loaded 1 modules   (3723 inline 8-bit counters): 3723 [0x67ccc0,
4      0x67db4b),
5  INFO: Loaded 1 PC tables (3723 PCs): 3723 [0x618d40,0x6275f0),
6  ./harness: Running 1 inputs 1 time(s) each.
7  Running: crash-7fd6f4dd5d39420d6f7887ff995b4e855ae90c16
8  =====
9  ==10973==ERROR: AddressSanitizer: heap-buffer-overflow on address
10 0x7bcece9e00a0 at pc 0x000000526c0e bp 0x7fff3dc0aa20 sp 0x7fff3dc0a1d8
11 WRITE of size 18 at 0x7bcece9e00a0 thread T0
12     #0 0x000000526c0d in strcpy
13         (/home/kchou/Bin/Repos/kchousos/OverHAuL/output/dateparse/harness
14         +0x526c0d) (BuildId: d658684b8726dc7e8e768089710d13c96cfc81f0)
15     #1 0x000000585555 in setFullMonth
16         /home/kchou/Bin/Repos/kchousos/OverHAuL/output/dateparse/harnesses
17         /harness.c:18:2
18     #2 0x0000005853fd in LLVMFuzzerTestOneInput
19         /home/kchou/Bin/Repos/kchousos/OverHAuL/output/dateparse/harnesses
20         /harness.c:41:2
21     ...
22 SUMMARY: AddressSanitizer: heap-buffer-overflow
23         (/home/kchou/Bin/Repos/kchousos/OverHAuL/output/dateparse/harness
24         +0x526c0d) (BuildId: d658684b8726dc7e8e768089710d13c96cfc81f0) in
25         strcpy
26 Shadow bytes around the buggy address:
27   0x7bcece9dfe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
28   0x7bcece9dfe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
29   0x7bcece9dff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30   0x7bcece9dff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
31   0x7bcece9e0000: fa fa 00 00 fa fa 00 fa fa fa 00 fa fa fa 00 fa
32   =>0x7bcece9e0080: fa fa 00 00[fa]fa fa fa fa fa fa fa fa fa fa fa
33   0x7bcece9e0100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
34   0x7bcece9e0180: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
35   0x7bcece9e0200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
36   0x7bcece9e0280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
37   0x7bcece9e0300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
38 Shadow byte legend (one shadow byte represents 8 application bytes):
```

Listing 3.8 Sample DSPy program.

```
1 import dspy
2 lm = dspy.LM('openai/gpt-4o-mini', api_key='YOUR_OPENAI_API_KEY')
3 dspy.configure(lm=lm)
4
5 math = dspy.ChainOfThought("question → answer: float")
6 math(question=(
7     "Two dice are tossed. What is the probability that the sum equals two?"
8 ))
```

4. Evaluation

To thoroughly assess the performance and effectiveness of OverHAuL, we established four *research questions* to direct our investigative efforts. These questions are designed to provide a structured framework for our inquiry and to ensure that our research remains focused on the key aspects of OverHAuL’s functionality and impact within its intended domain. By addressing these questions, we aim to uncover valuable insights that will contribute to a deeper understanding of OverHAuL’s capabilities and its position in contemporary automatic fuzzing applications:

- **RQ1:** Can OverHAuL generate working harnesses for unfuzzed C projects?
- **RQ2:** What characteristics do these harnesses have? Are they similar to man-made harnesses?
- **RQ3:** How do LLM usage patterns influence the generated harnesses?
- **RQ4:** How do different symbolic techniques affect the generated harnesses?

4.1. Experimental Benchmark

To evaluate OverHAuL, a benchmarking script was implemented¹ and a corpus of ten open-source C libraries was assembled. This collection comprises firstly of user dhvar’s “dateparse” library, which is also used as a running example in OSS-Fuzz-Gen’s [9] experimental from-scratch harnessing feature (Chapter 5). Secondly, nine other libraries chosen randomly² from the package catalog of Clib, a “package manager for the C programming language” [105], [106]. All libraries can be seen Table 4.1, along with their descriptions.

OverHAuL was evaluated through the experimental benchmark from 6th of June, 2025 to 18th of July, 2025, using OpenAI’s gpt-4.1-mini model [107]. For these runs, each OverHAuL execution was configured with a 5 minute harness execution timeout and an iteration budget of 10. Each benchmark run was executed as a GitHub Actions workflow on Linux virtual machines with 4-vCPUs and 16GiB of memory hosted on Microsoft Azure [108], [109]. The result directory (as described in Section 4.1.1) for each is available as a downloadable artifact in the corresponding GitHub Actions entry.

¹Available at <https://github.com/kchousos/OverHAuL/blob/master/benchmarks/benchmark.sh>.

²From the subset of libraries that do not have exotic external dependencies, like the X11 development toolchain.

Table 4.1.: The benchmark project corpus. Each project name links to its corresponding GitHub repository. Each is followed by a short description, its GitHub stars count and its Significant Lines of Code (SLOC), as of July 18th, 2025.

Project	Description	Stars	SLOC
dvhar/dateparse	A library that allows parsing dates without knowing the format in advance.	2	2272
clibs/buffer	A string manipulation library.	204	354
jwerle/libbeaufort	A library implementation of the Beaufort cipher [110].	13	321
jwerle/libbacon	A library implementation of the Baconian cipher [111].	8	191
jwerle/chfreq.c	A library for computing the character frequency in a string.	5	55
jwerle/progress.c	A library for displaying progress bars in the terminal.	76	357
willem/cbuffer	A circular buffer implementation.	261	170
willem/torrent-reader	A torrent-file reader library.	6	294
orangeduck/mpc	A type-generic parser combinator library.	2,753	3632
h2non/semver.c	A semantic version v2.0 parsing and rendering library [112].	190	608

4.1.1. Local Benchmarking

To run the benchmark locally, one would need to follow the installation instructions in Section 3.1 and then execute the benchmarking script, like so:

```
1 $ ./benchmarks/benchmark.sh
```

The cloned repositories with their corresponding harnesses will then be located in a subdirectory of `benchmark_results`, which will have the name format of `mini__<timestamp>__ReAct__<llm-model>__<max-exec-time>__<iter-budget>`. “Mini” corresponds to the benchmark project corpus described above, since a 30-project corpus was initially created and is now coined as “full” benchmark. Both the mini and full benchmarks are located in `benchmarks/repos.txt` and `benchmarks/repos-mini.txt` respectively. To execute the benchmark for the “full” corpus, users can add the `-b full` flag in the script’s invocation. Also, the LLM model used can be defined with the `-m` command-line flag.

4.2. Results

The outcomes of the benchmark experiments are shown in Figure 4.1. To ensure the reliability of these results, each reported crash was manually validated to confirm that it stemmed from genuine defects within the target library, rather than issues of the generated harness. An iteration heatmap was also generated for the verifiably fuzzed projects, displayed in Figure 4.2. With these validated findings, we are now positioned to address the initial research questions posed in this chapter.

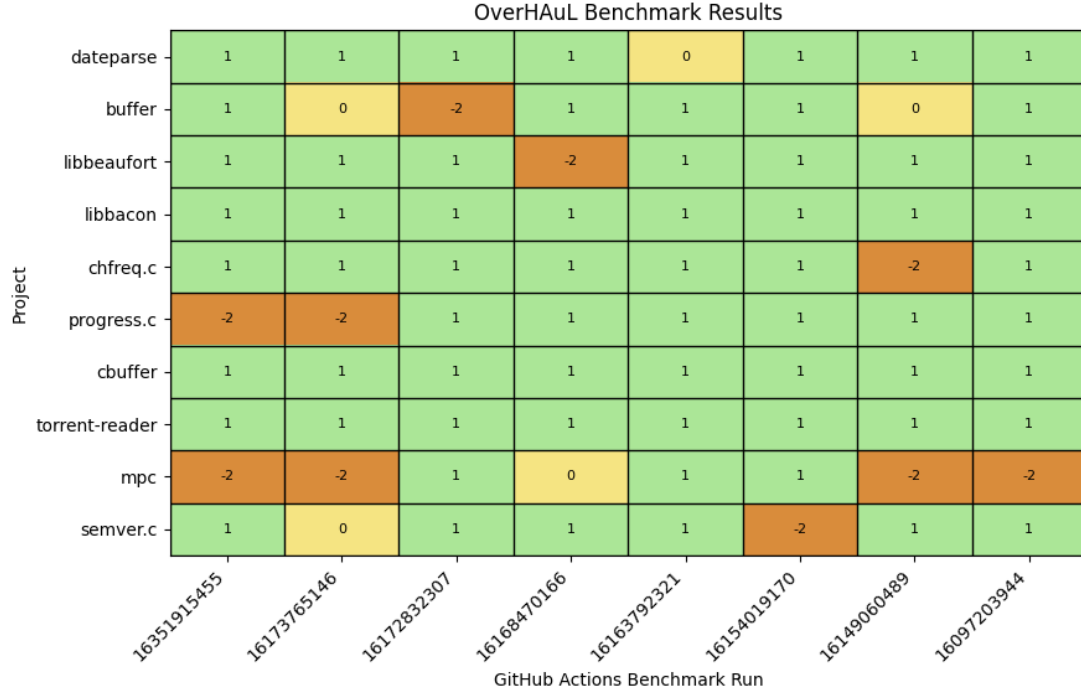


Figure 4.1.: The benchmark results for OverHAuL are illustrated with the y-axis depicting the ten-project corpus outlined in Section 4.1. The x-axis represents the various benchmark runs. Each label constitutes a unique hash identifier corresponding to a specific GitHub Actions workflow run, which can be accessed at <https://github.com/kchousos/OverHAuL/actions/runs/HASH>. An overview of all benchmark runs is available at <https://github.com/kchousos/OverHAuL/actions/workflows/benchmarks.yml>. In this matrix, a green/1 block indicates that OverHAuL successfully generated a new harness for the project and was able to find a crash input. On the other hand, a yellow/0 block indicates that while a compilable harness was produced, no crash input was found within the five-minute execution period. Finally, an orange/-2 block means that the crash that was found derives from errors in the harness itself. Alimportantly, there are no red/-1 blocks, which would indicate cases where a compilable harness could not be generated.

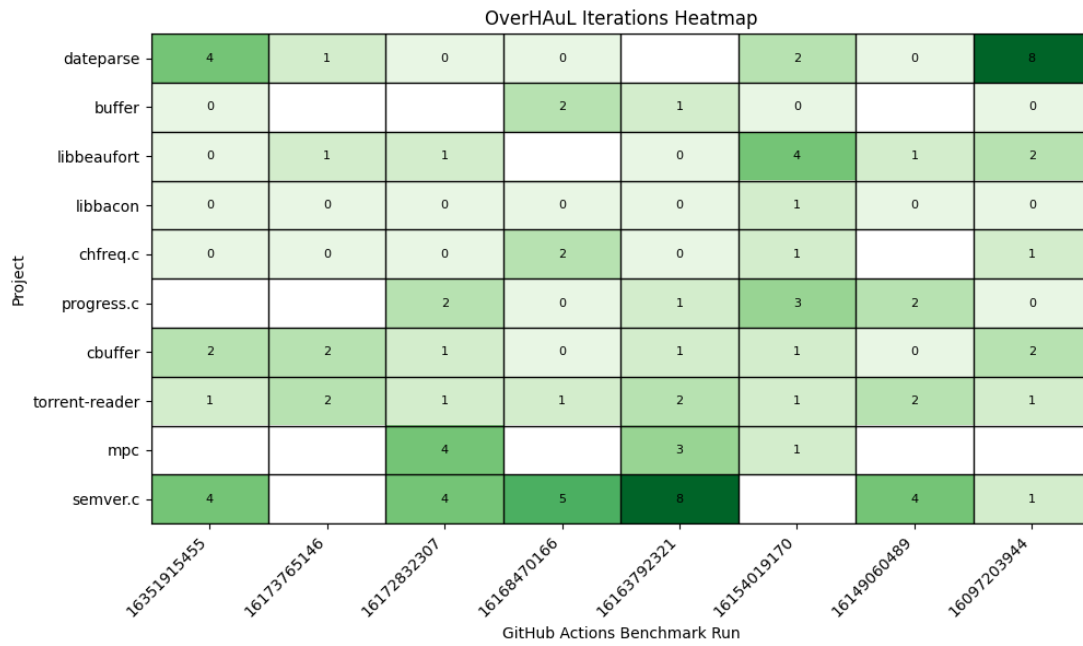


Figure 4.2.: This heatmap illustrates the number of iterations required for each project to be successfully harnessed, as determined by the benchmark results. Higher color intensity corresponds to a greater number of iterations needed for successful harnessing. Cells left blank signify instances where no valid harness was generated.

4.2.1. RQ 1: Can OverHAuL generate working harnesses for unfuzzed C projects?

OverHAuL demonstrates a strong capability in generating working harnesses for previously unfuzzed C projects. In benchmark evaluations, it achieved a success rate of **81.25%** in producing fuzzing harnesses that were effective at uncovering crash-inducing inputs in target programs. Notably, all harnesses generated by OverHAuL were valid C programs—an improvement over prior methods such as OSS-Fuzz-Gen [9], which occasionally outputs the LLM’s markdown answers instead. The harnesses consistently utilized existing functions obtained from the codebase oracle and interacted appropriately with the Library Under Test’s API, with only minimal instances of irrelevant or hallucinated code observed. While the potential exists for non-compilable harnesses to be generated, the benchmark results included no such cases, underscoring the significance and effectiveness of compilation feedback and the integrated “fixer” agent in OverHAuL’s workflow. These findings collectively indicate that OverHAuL is effective at generating robust, valid, and meaningful harnesses for C projects lacking previous fuzzing infrastructure.

4.2.2. RQ2: What characteristics do these harnesses have? Are they similar to man-made harnesses?

In examining the characteristics of the generated harnesses, we observe several notable patterns. The harnesses are typically well-commented, a result of explicit instructions given to the language models. They are designed to target various levels of the library’s functionality. In some cases, they focus on higher-level entry point functions (Section B.3), while in other instances, they concentrate on more narrowly scoped internal functions (Listing 3.5). Usually the generated fuzz targets are clear and closely resemble the kind of harnesses a skilled software engineer might write. These harnesses make appropriate and sensible use of the target API, as illustrated in examples such as Listing 3.5. However, some harnesses do exhibit the use of unexplained constants or idiosyncratic control flow constructs, which can hinder comprehensibility and may introduce errors. Additionally, we find that the characteristics of generated harnesses can vary substantially across different projects and even between runs, with differences evident in both their size and complexity (see Appendix B). Overall, while the generated harnesses often echo the structure and intent of man-made harnesses, inconsistencies and occasional inexplicable design choices distinguish them from their manually written counterparts.

4.2.3. RQ3: How do LLM usage patterns influence the generated harnesses?

The effectiveness of LLM-driven fuzzing harness generation in OverHAuL is heavily influenced by two primary factors: model selection and prompting strategies. The experimental evaluation presents compelling evidence regarding the substantial impact of both dimensions.

All benchmark experiments on GitHub’s infrastructure were conducted using OpenAI’s gpt-4.1-mini. Preliminary local testing included a spectrum of models—gpt-4.1, gpt-4o, gpt-4,

and gpt-3.5-turbo. Notably, both gpt-4.1 and gpt-4.1-mini achieved comparable performance, consistently generating robust fuzzing harnesses. In contrast, gpt-4o yielded somewhat average results, while gpt-4 and gpt-3.5-turbo exhibited significantly inferior performance, averaging only 2 out of 10 projects successfully harnessed per benchmark run. Models with suboptimal performance were excluded in subsequent development phases. These findings underscore the necessity of selecting advanced LLM architectures to realize OverHAuL’s potential; in particular, gpt-4o represents a recent baseline for acceptable performance. Because LLM model capabilities are evolving rapidly, it is reasonable to anticipate ongoing improvements in OverHAuL’s harness-generation efficacy as newer LLMs become available.

Prompting methodology is equally crucial. The adoption of ReAct prompting has proven most effective in the current implementation of OverHAuL [53]. Alternative prompting paradigms—including zero-shot and Chain-of-Thought (COT) approaches [50]—were empirically evaluated, as detailed in Appendix A, but failed to deliver satisfactory outcomes. A central challenge in automated harness generation involves ensuring that the resulting harness is both compilable and operationally effective. This alignment with real-world constraints necessitates continuous interaction between the LLM and the target environment, best achieved through agentic workflows [113]. The superior performance of ReAct prompting likely stems from its structured approach to iterative code exploration and refinement, facilitating a cycle of observation, planning, and action that is particularly well-suited to harness synthesis.

A central element of OverHAuL’s architecture is its triplet of ReAct agents, each contributing a distinct role in the collaborative generation of fuzzing harnesses. Local benchmarking demonstrates an almost linear increase in success rates with the number of iteration cycles, underscoring the efficacy of agentic collaboration and iterative refinement in enhancing harness quality. As illustrated in Figure 4.2, projects such as “dateparse” and “semver.c” exhibit marked improvements when afforded larger iteration budgets. This trend highlights the pivotal roles of the “fixer” and “improver” agents, whose interventions enable the system to surmount challenges present in initial harness generations, ultimately advancing the caliber of the final outputs.

Additionally, the inclusion of a codebase oracle is instrumental in scaling code exploration efficiently. Unlike previously tested methods (see Appendix A), the codebase oracle enables comprehensive traversal and understanding of project code, overcoming the token and context window limitations typically associated with LLMs.

In summary, the findings for RQ3 indicate that continuous advancements in LLM technology and prompting architectures will further enhance the ability of systems like OverHAuL to automate efficient fuzzing harness generation. Integrating agentic modules that can dynamically assess their environment and incorporate runtime feedback will likely outperform more static LLM applications, particularly within the domain of automated fuzzing.

4.2.4. RQ4: How do different symbolic techniques affect the generated harnesses?

Throughout the development of OverHAuL and its various iterations, numerous programming techniques were assessed in pursuit of answering RQ4 (Appendix A). Simple source code concatenation and its subsequent injection into LLM prompts revealed significant limitations, primarily due to the constraints of context windows. Conversely, the usage of tools capable of retrieving file contents marked a meaningful advancement. Nonetheless, this approach still encountered challenges, such as inaccessible code blocks and exploration that lacked semantic relevance. In response to these difficulties, the implementation of a function-level vector store functioning as a codebase oracle is proposed as a highly scalable solution. This strategy not only enhances the organization of larger files but also accommodates expanding project sizes, facilitating more semantically meaningful code examination.

The significance of the iterative feedback loop is clearly demonstrated by the results presented in Figure 4.2. Analysis of the heatmap reveals that earlier versions of OverHAuL, which employed a one-shot approach to harness generation, achieved a success rate of only 28.75%. In contrast, the current implementation shows that 42 out of 65 projects successfully fuzzed (64.62%) did not produce a successful harness in the initial attempt and therefore benefited from the iterative feedback process. Notably, two projects (3.07%) required the full allocation of eight iterations, underscoring the necessity of maintaining a generous iteration budget to maximize effectiveness.

4.3. Discussion

As discussed in Section 4.2, the capabilities and effectiveness of OverHAuL are closely tied to the choice of the underlying large language model. OverHAuL’s modular architecture ensures that advances in LLM research will directly enhance its performance. Each release of a new, more capable model can be readily integrated, thereby amplifying OverHAuL’s effectiveness without the need for substantial redesign.

A noteworthy consideration in our benchmarking setup is the possibility that some of the open-source libraries evaluated may have been included in the LLM’s training data. This introduces a risk of overestimating OverHAuL’s performance on code that is unseen or proprietary. Results for closed-source or less widely available libraries could therefore be weaker. Nonetheless, this potential limitation can theoretically be addressed through targeted fine-tuning of the LLM [114], [115].

4.3.1. Threats to Validity

Our evaluation of OverHAuL was conducted on ten relatively obscure open-source C libraries representing a range of application domains and functionalities. While this selection reduces

the likelihood that these projects were used in LLM training and thus minimizes potential bias, it remains uncertain how transferable our results are to larger, more complex, or structurally different codebases. Factors such as varying design paradigms, architectural patterns, or real-world deployment contexts may pose new challenges for OverHAuL’s scalability and effectiveness.

Additionally, the risk of LLM hallucination constitutes an internal threat to validity. Such hallucinations may require multiple attempts or occasional manual adjustments to produce valid and useful fuzz drivers. However, because LLMs—and thus OverHAuL—operate in a non-deterministic manner, it is possible to rerun the process and obtain alternative results. The inherent stochasticity of the underlying LLMs thus allows users to recover from initial failures, ensuring that the impact of hallucinations remains limited to efficiency rather than undermining the core applicability of the approach.

In summary, while our findings demonstrate the potential of OverHAuL, they also highlight important limitations and directions for future work, especially in improving robustness and evaluating performance across a broader spectrum of software projects.

5. Related work

Automated testing, automated fuzzing and automated harness creation have a long research history. Still, a lot of ground remains to be covered until true automation of these tasks is achieved. Until the introduction of transformers [39] and the 2020’s boom of commercial GPTs [48], automation regarding testing and fuzzing was mainly attempted through static and dynamic program analysis methods. These approaches are still utilized, but the fuzzing community has shifted almost entirely to researching the incorporation and employment of LLMs in the last half decade [9]–[12], [62], [65], [79], [116]–[118]. The most significant and recent works in this field can be categorized according to their primary methodologies—whether they employ program analysis techniques or LLMs—and by the extent to which they depend on external resources beyond the source code. It is important to note that these categories are not mutually exclusive.

5.1. Static and Dynamic Analysis-Powered Fuzzing

These tools employ both dynamic and static analyses of source code, as well as LLMs to enhance the automated generation of effective fuzz drivers.

KLEE [119] is a seminal and widely cited symbolic execution engine introduced in 2008 by Cadar et al. It was designed to automatically generate high-coverage test cases for programs written in C, using symbolic execution to systematically explore the control flow of a program. KLEE operates on the LLVM [33] bytecode representation of programs. Instead of executing a program on concrete inputs, KLEE performs symbolic execution—that is, it runs the program on symbolic inputs, which represent all possible values simultaneously. At each conditional branch, KLEE explores both paths by forking the execution and accumulating path constraints (i.e., logical conditions on input variables) along each path. This enables it to traverse many feasible execution paths in the program, including corner cases that may be difficult to reach through random testing or manual test creation. When an execution path reaches a terminal state (e.g., a program exit, an assertion failure, or a segmentation fault) KLEE uses a constraint solver to compute concrete input values that satisfy the accumulated constraints for that path. These values form a test case that will deterministically drive the program down that specific path when executed concretely.

IRIS [116] is a 2025 open-source neurosymbolic system for static vulnerability analysis. Given a codebase and a list of user-specified Common Weakness Enumerations (CWEs), it analyzes source code to identify paths that may correspond to known vulnerability classes. IRIS combines

symbolic analysis—such as control- and data-flow reasoning—with neural models trained to generalize over code patterns. It outputs candidate vulnerable paths along with explanations and CWE references. The system operates on full repositories and supports extensible CWE definitions.

IntelliGen [120] is a system for automatically synthesizing fuzz drivers by statically identifying potentially vulnerable entry-point functions within C projects. Implemented using LLVM [33], IntelliGen focuses on improving fuzzing efficiency by targeting code more likely to contain memory safety issues, rather than exhaustively fuzzing all available functions. The system comprises of two main components: the *Entry Function Locator* and the *Fuzz Driver Synthesizer*. The Entry Function Locator analyzes the project’s AST and classifies functions based on heuristics that indicate vulnerability. These include pointer dereferencing, calls to memory-related functions (e.g., `memcpy`, `memset`), and invocation of other internal functions. Functions that score highly on these metrics are prioritized for fuzz driver generation. The guiding insight is that entry points with fewer argument checks and more direct memory operations expose more useful program logic for fuzz testing. The Fuzz Driver Synthesizer then generates harnesses for these entry points. For each target function, it synthesizes an `LLVMFuzzerTestOneInput` function that invokes the target with arguments derived from the fuzz input. This process involves inferring argument types from the source code and ensuring that runtime behavior does not violate memory safety—thus avoiding invalid inputs that would cause crashes unrelated to genuine bugs.

CKGFuzzer [121] is a fuzzing framework designed to automate the generation of effective fuzz drivers for C/C++ libraries by leveraging static analysis and LLMs. Its workflow begins by parsing the target project along with any associated library APIs to construct a code knowledge graph. This involves two primary steps: first, parsing the AST, and second, performing inter-procedural program analysis. Through this process, CKGFuzzer extracts essential program elements such as function signatures and implementations, and call relationships. Using the knowledge graph, CKGFuzzer then identifies and queries meaningful API combinations, focusing on those that are either frequently invoked together or exhibit functional similarity. It generates candidate fuzz drivers for these combinations and attempts to compile them. Any compilation errors encountered are automatically repaired using heuristics and domain knowledge. A dynamically updated knowledge base, constructed from prior library usage patterns, guides both the generation and repair processes. Once the drivers are successfully compiled, CKGFuzzer executes them while monitoring code coverage. It uses coverage feedback to iteratively mutate underperforming API combinations, refining them until new execution paths are discovered or a preset mutation budget is exhausted. Finally, any crashes triggered during fuzzing are subjected to a reasoning process based on chain-of-thought prompting [50] (Section 2.2.2). To help determine their severity and root cause, CKGFuzzer consults an LLM-generated knowledge base containing real-world examples of vulnerabilities mapped to known CWE entries.

5.2. Extra Resources Required

The following works necessitate the presence of client code and/or unit tests that interact with the program’s API. These works utilize and modify such existing code to create enhanced fuzzing harnesses.

FUDGE [12] is a closed-source tool, made by Google, for automatic harness generation of C and C++ projects based on existing client code. It was used in conjunction with and in the improvement of Google’s OSS-Fuzz [80]. Being deployed inside Google’s infrastructure, FUDGE continuously examines Google’s internal code repository, searching for code that uses external libraries in a meaningful and “fuzzable” way (i.e. predominantly for parsing). If found, such code is *sliced* [122] based on its Abstract Syntax Tree (AST) using LLVM’s Clang tool [33]. The above process results in a set of abstracted mostly-self-contained code snippets that make use of a library’s calls and/or API. These snippets are later *synthesized* into the body of a fuzz driver, with variables being replaced and the fuzz input being utilized. Each is then injected in an `LLVMFuzzerTestOneInput` function and finalized as a fuzzing harness. A building and evaluation phase follows for each harness, where they are executed and examined. Every passing harness along with its evaluation results is stored in FUDGE’s database, reachable to the user through a custom web-based UI.

UTopia [10] (stylized UTOPIA) is an open-source automatic harness generation framework. Aside from the library code, It operates solely on user-provided unit tests since, according to Jeong et al. [10], they are a resource of complete and correct API usage examples containing working library set-ups and tear-downs. Additionally, each of them are already close to a fuzz target, in the sense that they already examine a single and self-contained API usage pattern. Each generated harness follows the same data flow of the originating unit test. Static analysis is employed to figure out what fuzz input placement would yield the most results. It is also utilized in abstracting the tests away from the syntactical differences between testing frameworks, along with slicing and AST traversing using Clang.

Another project of Google is FuzzGen [11], this time open-source. Like FUDGE, it leverages existing client code of the target library to create fuzz targets for it. FuzzGen uses whole-system analysis, through which it creates an *Abstract API Dependence Graph* (A^2DG). It uses the latter to automatically generate LibFuzzer-compatible harnesses. For FuzzGen to work, the user needs to provide both client code and/or tests for the API and the API library’s source code as well. FuzzGen uses the client code to infer the *correct usage* of the API and not its general structure, in contrast to FUDGE. FuzzGen’s workflow can be divided into three phases: 1. *API usage inference*. By analyzing client code and tests, FuzzGen recognizes which functions belong to the library and learns its correct API usage patterns. This process is done with the help of Clang. To test if a function is actually a part of the library, a sample program is created and compiled. If the program compiles successfully, then the function is indeed a valid API call. 2. *A^2DG construction mechanism*. For all the existing API calls, FuzzGen builds an A^2DG to record the API usages and infers its intended structure. After completion, this directed graph contains all the valid API call sequences found in the client code corpus. It is built in a two-step process: First, many smaller A^2DG s are created, one for each root function per client code snippet. Once

such graphs have been created for all the available client code instances, they are combined to formulate the master A²DG. 3. *Fuzzer generator*. Through the A²DG, a fuzzing harness is created. Contrary to FUDGE, FuzzGen does not create multiple “simple” harnesses but a single complex one with the goal of covering the whole A²DG.

5.3. Only Source Code Required

The approaches described in this section enable the creation of new fuzzing harnesses using exclusively the source code of the target library.

OSS-Fuzz [80], [123] is a continuous, scalable and distributed cloud fuzzing solution for critical and prominent open-source projects. Developers of such software can submit their projects to OSS-Fuzz’s platform, where its harnesses are built and constantly executed. This results in multiple bug findings that are later disclosed to the primary developers and are later patched. OSS-Fuzz started operating in 2016, an initiative in response to the Heartbleed vulnerability [22], [23], [25]. Its hope is that through more extensive fuzzing such errors could be caught and corrected before having the chance to be exploited and thus disrupt the public digital infrastructure. So far, it has helped uncover over 10,000 security vulnerabilities and 36,000 bugs across more than 1,000 projects, significantly enhancing the quality and security of major software like Chrome, OpenSSL, and Systemd. A project that’s part of OSS-Fuzz must have been configured as a ClusterFuzz [124] project. ClusterFuzz is the fuzzing infrastructure that OSS-Fuzz uses under the hood and depends on Google Cloud Platform services, although it is possible to host it locally. Such an integration requires setting up a build pipeline, fuzzing jobs and expects a Google Developer account. Results are accessible through a web interface. ClusterFuzz, and by extension OSS-Fuzz, supports fuzzing through LibFuzzer, AFL++, Honggfuzz and FuzzTest—successor to Centipede— with the last two being Google projects [19], [32], [125], [126]. C, C++, Rust, Go, Python and Java/JVM projects are supported.

OSS-Fuzz-Gen (OFG) [9], [127] is Google’s current state-of-the-art project regarding automatic harness generation through LLMs. Its purpose is to improve the fuzzing infrastructure of open-source projects that are already integrated into OSS-Fuzz. Given such a project, OSS-Fuzz-Gen uses its preexisting fuzzing harnesses and modifies them to produce new ones. Its architecture can be described as follows: 1. With an OSS-Fuzz project’s GitHub repository link, OSS-Fuzz-Gen iterates through a set of predefined build templates and generates potential build scripts for the project’s harnesses. 2. If any of them succeed they are once again compiled, this time through fuzz-introspector [81]. The latter constitutes a static analysis tool, with fuzzer developers specifically in mind. 3. Build results, old harness and fuzz-introspector report are included in a template-generated prompt, through which an LLM is called to generate a new harness. 4. The newly generated fuzz target is compiled and if it is done so successfully it begins execution inside OSS-Fuzz’s infrastructure. This method proves to be meaningful, with code coverage in fuzz campaigns increasing thanks to the new generated fuzz drivers. In the case of the tinycldr project [128], line coverage went from 38% to 69% without any manual interventions [127]. In 2024, OSS-Fuzz-Gen introduced an experimental feature for generating

harnesses in previously *unfuzzed* projects, meaning preexisting harnesses are no longer required [129]. Although this would be a step forward, this feature seems to have been abandoned. The code for this feature resides in the `experimental/from_scratch` directory of the project’s GitHub repository [9], with the latest known working commit being 171aac2 and the latest overall commit being four months ago, as of this writing.

AutoGen [79] is a closed-source tool that generates new fuzzing harnesses, given only the library code and documentation. The user specifies the function for which a harness is to be generated. AutoGen gathers information for this function—such as the function body, used header files, function calling examples—from the source code and documentation¹. Through specific prompt templates containing the above information, an LLM is tasked with generating a new fuzz driver, while another is tasked with generating a compilation command for said driver. If the compilation fails, both LLMs are called again to fix the problem, whether it was on the driver’s or command’s side. This loop iterates until a predefined maximum value or until a fuzz driver is successfully generated and compiled. If the latter is the case, it is then executed. If execution errors exist, the LLM responsible for the driver generation is used to correct them. If not, the pipeline has terminated and a new fuzz driver has been successfully generated.

5.4. Differences With OverHAuL

OverHAuL differs, in some way, with each of the aforementioned works. Firstly, although KLEE and IRIS [116], [119] tackle the problem of automated testing and both IRIS and OverHAuL can be considered neurosymbolic AI tools, the similarities end there. None of them utilize LLMs the same way we do—with KLEE not utilizing them at all, as it precedes them chronologically—and neither are automating any part of the fuzzing process.

When it comes to FUDGE, FuzzGen and UTopia [10]–[12], all three depend on and demand existing client code and/or unit tests. On the other hand, OverHAuL requires only the bare minimum: the library code itself. Another point of difference is that in contrast with OverHAuL, these tools operate in a linear fashion. No feedback is produced or used in any step and any point failure results in the termination of the entire run.

OverHAuL challenges a common principle of these tools, stated explicitly in FUDGE’s paper [12]: “Choosing a suitable fuzz target (still) requires a human”. OverHAuL chooses to let the LLM, instead of the user, explore the available functions and choose one to target in its fuzz driver.

Both IntelliGen and CKGFuzzer [120], [121] depend primarily on programmatic analysis of the target projects—like type inference and knowledge graph construction, respectively. In contrast, OverHAuL delegates a greater portion of this analytical workload to LLM agents, leveraging their reasoning capabilities to achieve more accurate and reliable outcomes.

¹Therefore, while no pre-existing client code is necessary, available documentation remains essential.

OSS-Fuzz-Gen [9] can be considered a close counterpart of OverHAuL, and in some ways it is. A lot of inspiration was gathered from it, like for example the inclusion of static analysis and its usage in informing the LLM. Yet, OSS-Fuzz-Gen has a number of disadvantages that make it in some cases an inferior option. For one, OFG is tightly coupled with the OSS-Fuzz platform [80], which even on its own creates a plethora of issues for the common developer. To integrate their project into OSS-Fuzz, they would need to: Transform it into a ClusterFuzz project [124] and take time to write harnesses for it. Even if these prerequisites are carried out, it probably would not be enough. Per OSS-Fuzz’s documentation [123]: “To be accepted to OSS-Fuzz, an open-source project must have a significant user base and/or be critical to the global IT infrastructure”. This means that OSS-Fuzz is a viable option only for a small minority of open-source developers and maintainers. One countermeasure of the above shortcoming would be for a developer to run OSS-Fuzz-Gen locally. This unfortunately proves to be an arduous task. As it is not meant to be used standalone, OFG is not packaged in the form of a self-contained application. This makes it hard to setup and difficult to use interactively. Like in the case of FUDGE, OFG’s actions are performed linearly. No feedback is utilized nor is there graceful error handling in the case of a step’s failure. Even in the case of the experimental feature for bootstrapping unfuzzed projects, OFG’s performance varies heavily. During experimentation, a lot of generated harnesses were still wrapped either in Markdown backticks or `tags, or were accompanied with explanations inside the generated .c source file. Even if code was formatted correctly, in many cases it missed necessary headers for compilation or used undeclared functions.`

Lastly, the closest counterpart to OverHAuL is AutoGen [79]. Their similarity stands in the implementation of a feedback loop between LLM and generated harness. However, most other implementation decisions remain distinct. One difference regards the fuzzed function. While AutoGen requires a target function to be specified by the user in which it narrows during its whole run, OverHAuL delegates this to the LLM, letting it explore the codebase and decide by itself the best candidate. Another difference lies in the need—and the lack of—of documentation. While AutoGen requires it to gather information for the given function, OverHAuL leans into the role of a developer by reading the related code and comments and thus avoiding any mismatches between documentation and code. Finally, the LLMs’ input is built based on predefined prompt templates, a technique also present in OSS-Fuzz-Gen. OverHAuL operates one abstraction level higher, leveraging DSPy [92] for programming instead of prompting the LLMs used.

In conclusion, OverHAuL constitutes an *open-source* tool that offers new functionality by offering a straightforward installation process, packaged as a self-contained Python package with minimal external dependencies. It also introduces novel approaches compared to previous work by

1. Implementing a feedback mechanism between harness generation, compilation, and evaluation phases,
2. Using autonomous ReAct agents capable of codebase exploration,
3. Leveraging a vector store for code consumption and retrieval.

6. Future Work

The prototype implementation of OverHAuL offers a compelling demonstration of its potential to automate the fuzzing process for open-source libraries, providing tangible benefits to developers and maintainers alike. This initial version successfully validates the core design principles underpinning OverHAuL, showcasing its ability to streamline and enhance the software testing workflow through automated generation of fuzz drivers using large language models. Nevertheless, while these foundational capabilities lay a solid groundwork, numerous avenues exist for further expansion, refinement, and rigorous evaluation to fully realize the tool’s potential and adapt to evolving challenges in software quality assurance.

6.1. Enhancements to Core Features

Enhancing OverHAuL’s core functionality represents a primary direction for future development. First, expanding support to encompass a wider array of build systems commonly employed in C and C++ projects—such as GNU Make, CMake, Meson, and Ninja [87], [88], [130], [131]—would significantly broaden the scope of libraries amenable to automated fuzzing using OverHAuL. This advancement would enable OverHAuL to scale effectively and be applied to larger, more complex codebases, thereby increasing its practical utility and impact.

Second, integrating additional fuzzing engines beyond LibFuzzer stands out as a strategic enhancement. Incorporation of widely adopted fuzzers like AFL++ [32] could diversify the fuzzing strategies available to OverHAuL, while exploring more experimental tools such as GraphFuzz [118] may pioneer specialized approaches for certain code patterns or architectures. Multi-engine support would also facilitate extending language coverage, for instance by incorporating fuzzers tailored to other programming ecosystems—for example, Google’s Atheris for Python projects [132]. Such versatility would position OverHAuL as a more universal fuzzing automation platform.

Third, the evaluation component of OverHAuL presents an opportunity for refinement through more sophisticated analysis techniques. Beyond the current criteria, incorporating dynamic metrics such as differential code coverage tracking between generated fuzz harnesses would yield deeper insights into test quality and coverage completeness. This quantitative evaluation could guide iterative improvements in fuzz driver generation and overall testing effectiveness.

Finally, OverHAuL’s methodology could be extended to leverage existing client codebases and unit tests in addition to the library source code itself, resources that for now OverHAuL leaves untapped. Inspired by approaches like those found in FUDGE and FuzzGen [11], [12], this

enhancement would enable the tool to exploit programmer-written usage scenarios as seeds or contexts, potentially generating more meaningful and targeted fuzz inputs. Incorporating these richer information sources would likely improve the efficacy of fuzzing campaigns and uncover subtler bugs.

6.2. Experimentation with Large Language Models and Data Representation

OverHAuL’s reliance on large language models (LLMs) invites comprehensive experimentation with different providers and architectures to assess their comparative strengths and limitations. Conducting empirical evaluations across leading models—such as OpenAI’s o1 and o3 families and Anthropic’s Claude Opus 4—will provide valuable insights into their capabilities, cost-efficiency, and suitability for fuzz driver synthesis. Additionally, specialized code-focused LLMs, including generative and fill-in models like Codex-1 and CodeGen [56]–[58], merit exploration due to their targeted optimization for source code generation and understanding.

Another dimension worthy of investigation concerns the granularity of code chunking employed during the given project’s code processing stage. Whereas the current approach partitions code at the function level, experimenting with more nuanced segmentation strategies—such as splitting per step inside a function, as a finer-grained technique—could improve the semantic coherence of stored representations and enhance retrieval relevance during fuzz driver generation. This line of inquiry has the potential to optimize model input preparation and ultimately improve output quality.

6.3. Comprehensive Evaluation and Benchmarking

To thoroughly establish OverHAuL’s effectiveness, extensive large-scale evaluation beyond the initial 10-project corpus is imperative. Applying the tool to repositories indexed in the `clib` package manager [105], which encompasses hundreds of C libraries, would test scalability and robustness across diverse real-world settings. Such a broad benchmark would also enable systematic comparisons against state-of-the-art automated fuzzing frameworks like OSS-Fuzz-Gen and AutoGen, elucidating OverHAuL’s relative strengths and identifying areas for improvement [9], [79].

Complementing broad benchmarking, detailed ablation and matrix studies dissecting the contributions of individual pipeline components and algorithmic choices will yield critical insights into what drives OverHAuL’s performance. Understanding the impact of each module will guide targeted optimizations and support evidence-based design decisions.

Furthermore, an economic analysis exploring resource consumption—such as API token usage and associated monetary costs—relative to fuzzing effectiveness would be valuable for assess-

ing the practical viability of integrating LLM-based fuzz driver generation into continuous integration processes.

6.4. Practical Deployment and Community Engagement

From a usability perspective, embedding OverHAuL within a GitHub Actions workflow represents a practical and impactful enhancement, enabling seamless integration with developers' existing toolchains and continuous integration pipelines. This would promote wider adoption by reducing barriers to entry and fostering real-time feedback during code development cycles.

Additionally, establishing a mechanism to generate and submit automated pull requests (PRs) to the maintainers of fuzzed libraries—highlighting detected bugs and proposing patches—would not only validate OverHAuL's findings but also contribute tangible improvements to open-source software quality. This collaborative feedback loop epitomizes the symbiosis between automated testing tools and the open-source community. As an initial step, developing targeted PRs for the projects where bugs were discovered during OverHAuL's development would help facilitate practical follow-up and improvements.

7. Conclusion

This thesis set out to address a pressing challenge in software testing for legacy and under-tested C codebases: the significant manual effort required to develop fuzzing harnesses, especially in the absence of pre-existing test infrastructure. In response, we present OverHAuL, a neurosymbolic AI system capable of autonomously generating effective fuzzing harnesses directly from source code. OverHAuL leverages the strengths of advanced large language model (LLM) agents, enabling it to overcome the traditional dependencies on manual effort, client code, or existing test harnesses that characterize previous tools.

Central to OverHAuL’s methodology is the integration of a triplet of ReAct LLM agents working within a feedback-oriented, iterative loop, capable of investigating the given project’s source code through a codebase oracle. This architecture allows the system to intelligently explore otherwise opaque codebases, systematically identifying candidate entry points for fuzzing and synthesizing robust harnesses. The end-to-end automation pipeline incorporates a compilation and evaluation phase, during which the generated harnesses are systematically compiled and rigorously assessed for correctness and effectiveness.

To rigorously assess OverHAuL’s efficacy and reliability, we designed a comprehensive evaluation using a benchmark suite of ten open-source C libraries. Our experiments demonstrate that OverHAuL successfully produced valid and usable fuzzing harnesses in 81.25% of the cases. This high success rate offers strong evidence supporting OverHAuL’s correctness and practical applicability, substantiating the central hypothesis of this thesis.

Through a comprehensive literature review of prominent related projects and a detailed comparative analysis between them and OverHAuL, we demonstrate that OverHAuL distinguishes itself in several critical aspects. Our system’s high degree of automation and limited dependence on external artifacts constitute significant advantages over previous methods, particularly regarding its applicability to legacy or inadequately documented C codebases. OverHAuL’s novel methodology underscores its distinctive role within the rapidly evolving landscape of automated fuzzing solutions, especially when contrasted against other state-of-the-art approaches.

Looking ahead, this body of work invites several promising directions for future exploration. Expanding OverHAuL’s applicability to additional programming languages and improving compatibility with established build ecosystems would significantly widen its practical impact. Ongoing refinements to its AI-driven algorithms, especially in areas of program slicing and harness evaluation, have the potential to further enhance the robustness and effectiveness of the system. Lastly, conducting more comprehensive evaluations and large-scale comparisons with state-of-the-art tools would provide stronger evidence for the effectiveness of OverHAuL, further demonstrating its superiority over existing solutions.

In summary, this thesis advances the field of automated software testing by demonstrating the feasibility and utility of autonomously generated fuzzing harnesses for C projects. OverHAuL establishes a compelling foundation for future research, representing a substantial step towards fully automated, scalable, and intelligent fuzzing infrastructure in the face of increasingly complex software systems.

Bibliography

- [1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice-Hall Software Series). Englewood Cliffs, N.J.: Prentice-Hall, 1978, 228 pp., ISBN: 978-0-13-110163-0.
- [2] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, “The C programming language,” *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1991–2019, 1978. [Online]. Available: https://www.academia.edu/download/67840358/1978.07_Bell_System_Technical_Journal.pdf#page=85.
- [3] G. J. Holzmann, “The Power of 10: Rules for Developing Safety-Critical Code,” Jun. 2006. [Online]. Available: <https://web.eecs.umich.edu/~imarkov/10rules.pdf>.
- [4] Ada Developers. “Ada Reference Manual, 2022 Edition,” Ada Information Clearinghouse. (2022), [Online]. Available: https://www.adaic.org/resources/add_content/standards/22rm/html/RM-TTL.html.
- [5] Rust Project Developers. “Rust Programming Language.” (2025), [Online]. Available: <https://www.rust-lang.org/>.
- [6] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. “Do Users Write More Insecure Code with AI Assistants?” arXiv: [2211.03622](https://arxiv.org/abs/2211.03622). (Dec. 18, 2023), [Online]. Available: <http://arxiv.org/abs/2211.03622>, pre-published.
- [7] N. Kosmyrna, E. Hauptmann, Y. T. Yuan, *et al.* “Your Brain on ChatGPT: Accumulation of Cognitive Debt when Using an AI Assistant for Essay Writing Task.” arXiv: [2506.08872](https://arxiv.org/abs/2506.08872) [cs]. (Jun. 10, 2025), [Online]. Available: <http://arxiv.org/abs/2506.08872>, pre-published.
- [8] H.-P. H. Lee, A. Sarkar, L. Tankelevitch, *et al.*, “The Impact of Generative AI on Critical Thinking: Self-Reported Reductions in Cognitive Effort and Confidence Effects From a Survey of Knowledge Workers,” 2025. [Online]. Available: https://hankhplee.com/papers/genai_critical_thinking.pdf.
- [9] D. Liu, O. Chang, J. metzman, M. Sablotny, and M. Maruseac, *OSS-fuzz-gen: Automated fuzz target generation*, version <https://github.com/google/oss-fuzz-gen/tree/v1.0>, May 2024. [Online]. Available: <https://github.com/google/oss-fuzz-gen>.
- [10] B. Jeong, J. Jang, H. Yi, *et al.*, “UTopia: Automatic Generation of Fuzz Driver using Unit Tests,” in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2676–2692. doi: [10.1109/SP46215.2023.10179394](https://doi.org/10.1109/SP46215.2023.10179394). [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10179394>.
- [11] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, “FuzzGen: Automatic fuzzer generation,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou>.
- [12] D. Babić, S. Bucur, Y. Chen, *et al.*, “FUDGE: Fuzz driver generation at scale,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 12, 2019, pp. 975–985, ISBN: 978-1-4503-5572-8. doi: [10.1145/3338906.3340456](https://doi.org/10.1145/3338906.3340456). [Online]. Available: <https://dl.acm.org/doi/10.1145/3338906.3340456>.

- [13] V. J. M. Manes, H. Han, C. Han, *et al.* “The Art, Science, and Engineering of Fuzzing: A Survey.” arXiv: [1812.00140 \[cs\]](https://arxiv.org/abs/1812.00140). (Apr. 7, 2019), [Online]. Available: <http://arxiv.org/abs/1812.00140>, pre-published.
- [14] A. Takanen, J. DeMott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance* (Information Security and Privacy Library), Second edition. Boston London Norwood, MA: Artech House, 2018, 1 p., ISBN: 978-1-63081-519-6.
- [15] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Upper Saddle River, NJ: Addison-Wesley, 2007, 543 pp., ISBN: 978-0-321-44611-4.
- [16] N. Rathaus and G. Evron, *Open Source Fuzzing Tools*, G. Evron, Ed. Burlington, MA: Syngress Pub, 2007, 199 pp., ISBN: 978-1-59749-195-2.
- [17] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1, 1990, ISSN: 0001-0782. DOI: [10.1145/96267.96279](https://doi.org/10.1145/96267.96279). [Online]. Available: <https://dl.acm.org/doi/10.1145/96267.96279>.
- [18] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318. [Online]. Available: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>.
- [19] LLVM Project. “libFuzzer – a library for coverage-guided fuzz testing. — LLVM 21.0.0git documentation.” (2025), [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>.
- [20] A. Rebert, S. K. Cha, T. Avgerinos, *et al.*, “Optimizing seed selection for fuzzing,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14, USA: USENIX Association, Aug. 20, 2014, pp. 861–875, ISBN: 978-1-931971-15-7.
- [21] OWASP Foundation. “Fuzzing.” (), [Online]. Available: <https://owasp.org/www-community/Fuzzing>.
- [22] Blackduck, Inc. “Heartbleed Bug.” (Mar. 7, 2025), [Online]. Available: <https://heartbleed.com/>.
- [23] CVE Program. “CVE - CVE-2014-0160.” (2014), [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160>.
- [24] The OpenSSL Project, *Openssl/openssl*, OpenSSL, Jul. 15, 2025. [Online]. Available: <https://github.com/openssl/openssl>.
- [25] D. Wheeler. “How to Prevent the next Heartbleed.” (2014), [Online]. Available: <https://dwheeler.com/essays/heartbleed.html>.
- [26] GNU Project. “Bash - GNU Project - Free Software Foundation.” (), [Online]. Available: <https://www.gnu.org/software/bash/>.
- [27] M. Zalewski. “American fuzzy lop.” (), [Online]. Available: <https://lcamtuf.coredump.cx/afl/>.
- [28] J. Saarinen. “Further flaws render Shellshock patch ineffective,” *iTnews*. (Sep. 29, 2014), [Online]. Available: <https://www.itnews.com.au/news/further-flaws-render-shellshock-patch-ineffective-396256>.
- [29] T. Avgerinos, D. Brumley, J. Davis, *et al.*, “The mayhem cyber reasoning system,” *IEEE Security & Privacy*, vol. 16, no. 2, pp. 52–60, 2018.
- [30] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *2012 IEEE Symposium on Security and Privacy*, IEEE, 2012, pp. 380–394.
- [31] T. Simonite, “This Bot Hunts Software Bugs for the Pentagon,” *Wired*, Jun. 1, 2020, ISSN: 1059-1028. [Online]. Available: <https://www.wired.com/story/bot-hunts-software-bugs-pentagon/>.

- [32] M. Heuse, H. Eißfeldt, A. Fioraldi, and D. Maier, *AFL++*, version 4.00c, Jan. 2022. [Online]. Available: <https://github.com/AFLplusplus/AFLplusplus>.
- [33] LLVM Project. “The LLVM Compiler Infrastructure Project.” (2025), [Online]. Available: <https://llvm.org/>.
- [34] F. Bellard, P. Maydell, and QEMU Team, *QEMU*, version 10.0.2, May 29, 2025. [Online]. Available: <https://www.qemu.org/>.
- [35] Unicorn Engine, *Unicorn-engine/unicorn*, Unicorn Engine, Jul. 15, 2025. [Online]. Available: <https://github.com/unicorn-engine/unicorn>.
- [36] H. Li, “Language models: Past, present, and future,” *Commun. ACM*, vol. 65, no. 7, pp. 56–63, Jun. 21, 2022, issn: 0001-0782. doi: [10.1145/3490443](https://doi.org/10.1145/3490443). [Online]. Available: <https://dl.acm.org/doi/10.1145/3490443>.
- [37] Z. Wang, Z. Chu, T. V. Doan, S. Ni, M. Yang, and W. Zhang, “History, development, and principles of large language models: An introductory survey,” *AI and Ethics*, vol. 5, no. 3, pp. 1955–1971, Jun. 1, 2025, issn: 2730-5961. doi: [10.1007/s43681-024-00583-7](https://doi.org/10.1007/s43681-024-00583-7). [Online]. Available: <https://doi.org/10.1007/s43681-024-00583-7>.
- [38] D. Bahdanau, K. Cho, and Y. Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate.” arXiv: [1409.0473](https://arxiv.org/abs/1409.0473) [cs, stat]. (May 19, 2016), [Online]. Available: <http://arxiv.org/abs/1409.0473>, pre-published.
- [39] A. Vaswani, N. Shazeer, N. Parmar, *et al.* “Attention Is All You Need.” arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs]. (Aug. 1, 2023), [Online]. Available: <http://arxiv.org/abs/1706.03762>, pre-published.
- [40] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” arXiv: [1810.04805](https://arxiv.org/abs/1810.04805) [cs]. (May 24, 2019), [Online]. Available: <http://arxiv.org/abs/1810.04805>, pre-published.
- [41] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” 2018. [Online]. Available: <https://www.mikecaptain.com/resources/pdf/GPT-1.pdf>.
- [42] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019. [Online]. Available: <https://storage.prod.researchhub.com/uploads/papers/2020/06/01/language-models.pdf>.
- [43] T. B. Brown, B. Mann, N. Ryder, *et al.* “Language Models are Few-Shot Learners.” arXiv: [2005.14165](https://arxiv.org/abs/2005.14165) [cs]. (Jul. 22, 2020), [Online]. Available: <http://arxiv.org/abs/2005.14165>, pre-published.
- [44] OpenAI, J. Achiam, S. Adler, *et al.* “GPT-4 Technical Report.” arXiv: [2303.08774](https://arxiv.org/abs/2303.08774) [cs]. (Mar. 4, 2024), [Online]. Available: <http://arxiv.org/abs/2303.08774>, pre-published.
- [45] Anthropic. “Claude.” (2025), [Online]. Available: <https://claude.ai/new>.
- [46] DeepSeek-AI, D. Guo, D. Yang, *et al.* “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.” arXiv: [2501.12948](https://arxiv.org/abs/2501.12948) [cs]. (Jan. 22, 2025), [Online]. Available: <http://arxiv.org/abs/2501.12948>, pre-published.
- [47] A. Grattafiori, A. Dubey, A. Jauhri, *et al.* “The Llama 3 Herd of Models.” arXiv: [2407.21783](https://arxiv.org/abs/2407.21783) [cs]. (Nov. 23, 2024), [Online]. Available: <http://arxiv.org/abs/2407.21783>, pre-published.
- [48] OpenAI. “ChatGPT.” (2025), [Online]. Available: <https://chatgpt.com>.
- [49] Google. “Google Gemini,” Gemini. (2025), [Online]. Available: <https://gemini.google.com>.
- [50] J. Wei, X. Wang, D. Schuurmans, *et al.* “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.” arXiv: [2201.11903](https://arxiv.org/abs/2201.11903) [cs]. (Jan. 10, 2023), [Online]. Available: <http://arxiv.org/abs/2201.11903>, pre-published.

- [51] S. Yao, D. Yu, J. Zhao, *et al.* “Tree of Thoughts: Deliberate Problem Solving with Large Language Models.” arXiv: [2305.10601](https://arxiv.org/abs/2305.10601) [cs]. (Dec. 3, 2023), [Online]. Available: [http://arxiv.org/abs/2305.10601](https://arxiv.org/abs/2305.10601), pre-published.
- [52] P. Lewis, E. Perez, A. Piktus, *et al.* “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” arXiv: [2005.11401](https://arxiv.org/abs/2005.11401) [cs]. (Apr. 12, 2021), [Online]. Available: [http://arxiv.org/abs/2005.11401](https://arxiv.org/abs/2005.11401), pre-published.
- [53] S. Yao, J. Zhao, D. Yu, *et al.* “ReAct: Synergizing Reasoning and Acting in Language Models.” arXiv: [2210.03629](https://arxiv.org/abs/2210.03629). (Mar. 10, 2023), [Online]. Available: [http://arxiv.org/abs/2210.03629](https://arxiv.org/abs/2210.03629), pre-published.
- [54] Anysphere. “Cursor - The AI Code Editor.” (2025), [Online]. Available: <https://cursor.com/>.
- [55] Microsoft. “GitHub Copilot · Your AI pair programmer,” GitHub. (2025), [Online]. Available: <https://github.com/features/copilot>.
- [56] E. Nijkamp, B. Pang, H. Hayashi, *et al.*, “CodeGen: An open large language model for code with multi-turn program synthesis,” *ICLR*, 2023.
- [57] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, “CodeGen2: Lessons for training llms on programming and natural languages,” *ICLR*, 2023.
- [58] OpenAI. “Introducing GPT-4.1 in the API.” (Apr. 14, 2025), [Online]. Available: <https://openai.com/index/gpt-4-1/>.
- [59] A. Sarkar and I. Drosos. “Vibe coding: Programming through conversation with artificial intelligence.” arXiv: [2506.23253](https://arxiv.org/abs/2506.23253) [cs]. (Jun. 29, 2025), [Online]. Available: [http://arxiv.org/abs/2506.23253](https://arxiv.org/abs/2506.23253), pre-published.
- [60] L. Huang, W. Yu, W. Ma, *et al.*, “A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions,” *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, Mar. 31, 2025, ISSN: 1046-8188, 1558-2868. DOI: [10.1145/3703155](https://doi.org/10.1145/3703155). arXiv: [2311.05232](https://arxiv.org/abs/2311.05232) [cs]. [Online]. Available: [http://arxiv.org/abs/2311.05232](https://arxiv.org/abs/2311.05232).
- [61] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy. “Challenges and Applications of Large Language Models.” arXiv: [2307.10169](https://arxiv.org/abs/2307.10169) [cs]. (Jul. 19, 2023), [Online]. Available: [http://arxiv.org/abs/2307.10169](https://arxiv.org/abs/2307.10169), pre-published.
- [62] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “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*, ser. ISSTA 2023, New York, NY, USA: Association for Computing Machinery, Jul. 13, 2023, pp. 423–435, ISBN: 979-8-4007-0221-1. DOI: [10.1145/3597926.3598067](https://doi.org/10.1145/3597926.3598067). [Online]. Available: <https://dl.acm.org/doi/10.1145/3597926.3598067>.
- [63] G. Black, V. Mathew Vaidyan, and G. Comert, “Evaluating Large Language Models for Enhanced Fuzzing: An Analysis Framework for LLM-Driven Seed Generation,” *IEEE Access*, vol. 12, pp. 156 065–156 081, 2024, ISSN: 2169-3536. DOI: [10.1109/ACCESS.2024.3484947](https://doi.org/10.1109/ACCESS.2024.3484947). [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10731701>.
- [64] W. Shi, Y. Zhang, X. Xing, and J. Xu. “Harnessing Large Language Models for Seed Generation in Greybox Fuzzing.” arXiv: [2411.18143](https://arxiv.org/abs/2411.18143) [cs]. (Nov. 27, 2024), [Online]. Available: [http://arxiv.org/abs/2411.18143](https://arxiv.org/abs/2411.18143), pre-published.
- [65] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang. “Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT.” arXiv: [2304.02014](https://arxiv.org/abs/2304.02014) [cs]. (Apr. 4, 2023), [Online]. Available: [http://arxiv.org/abs/2304.02014](https://arxiv.org/abs/2304.02014), pre-published.

- [66] Y. Jiang, J. Liang, F. Ma, *et al.*, “When Fuzzing Meets LLMs: Challenges and Opportunities,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. ACM Conferences, Jul. 10, 2024, pp. 492–496, ISBN: 979-8-4007-0658-5. DOI: [10.1145/3663529.3663784](https://doi.org/10.1145/3663529.3663784). [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/3663529.3663784>.
- [67] D. Tilwani, R. Venkataramanan, and A. P. Sheth. “Neurosymbolic AI approach to Attribution in Large Language Models.” arXiv: [2410.03726](https://arxiv.org/abs/2410.03726). (Sep. 30, 2024), [Online]. Available: <http://arxiv.org/abs/2410.03726>, pre-published.
- [68] D. Kahneman, *Thinking, Fast and Slow*, 1st ed. New York: Farrar, Straus and Giroux, 2011, 499 pp., ISBN: 978-0-374-27563-1 978-0-374-53355-7 978-0-606-27564-4.
- [69] A. Mastropaolo and D. Poshyvanyk. “A Path Less Traveled: Reimagining Software Engineering Automation via a Neurosymbolic Paradigm.” arXiv: [2505.02275](https://arxiv.org/abs/2505.02275) [cs]. (May 4, 2025), [Online]. Available: <http://arxiv.org/abs/2505.02275>, pre-published.
- [70] A. Velasco, A. Garryeva, D. N. Palacio, A. Mastropaolo, and D. Poshyvanyk. “Toward Neurosymbolic Program Comprehension.” arXiv: [2502.01806](https://arxiv.org/abs/2502.01806) [cs]. (Feb. 3, 2025), [Online]. Available: <http://arxiv.org/abs/2502.01806>, pre-published.
- [71] A. Sheth, K. Roy, and M. Gaur. “Neurosymbolic AI – Why, What, and How.” arXiv: [2305.00813](https://arxiv.org/abs/2305.00813) [cs]. (May 1, 2023), [Online]. Available: <http://arxiv.org/abs/2305.00813>, pre-published.
- [72] A. d’Avila Garcez and L. C. Lamb. “Neurosymbolic AI: The 3rd Wave.” arXiv: [2012.05876](https://arxiv.org/abs/2012.05876). (Dec. 16, 2020), [Online]. Available: <http://arxiv.org/abs/2012.05876>, pre-published.
- [73] D. Ganguly, S. Iyengar, V. Chaudhary, and S. Kalyanaraman. “Proof of Thought : Neurosymbolic Program Synthesis allows Robust and Interpretable Reasoning.” arXiv: [2409.17270](https://arxiv.org/abs/2409.17270). (Sep. 25, 2024), [Online]. Available: <http://arxiv.org/abs/2409.17270>, pre-published.
- [74] M. Gaur and A. Sheth. “Building Trustworthy NeuroSymbolic AI Systems: Consistency, Reliability, Explainability, and Safety.” arXiv: [2312.06798](https://arxiv.org/abs/2312.06798). (Dec. 5, 2023), [Online]. Available: <http://arxiv.org/abs/2312.06798>, pre-published.
- [75] M. K. Sarker, L. Zhou, A. Eberhart, and P. Hitzler, “Neuro-symbolic artificial intelligence: Current trends,” *AI Communications*, vol. 34, no. 3, pp. 197–209, Mar. 4, 2022, ISSN: 1875-8452, 0921-7126. DOI: [10.3233/aic-210084](https://doi.org/10.3233/aic-210084). [Online]. Available: <https://journals.sagepub.com/doi/full/10.3233/AIC-210084>.
- [76] H. Kautz, “The Third AI Summer,” Lecture, presented at the 34th Annual Meeting of the Association for the Advancement of Artificial Intelligence (New York, NY, USA), Feb. 10, 2020. [Online]. Available: https://www.youtube.com/watch?v=_cQITY0SPiw.
- [77] L. Torvalds, *Git*, Apr. 7, 2005. [Online]. Available: <https://git-scm.com/>.
- [78] T. Mikolov, K. Chen, G. Corrado, and J. Dean. “Efficient Estimation of Word Representations in Vector Space.” arXiv: [1301.3781](https://arxiv.org/abs/1301.3781) [cs]. (Sep. 6, 2013), [Online]. Available: <http://arxiv.org/abs/1301.3781>, pre-published.
- [79] Y. Sun, “Automated Generation and Compilation of Fuzz Driver Based on Large Language Models,” in *Proceedings of the 2024 9th International Conference on Cyber Security and Information Engineering*, ser. ICCSIE ’24, New York, NY, USA: Association for Computing Machinery, Dec. 3, 2024, pp. 461–468, ISBN: 979-8-4007-1813-7. DOI: [10.1145/3689236.3689272](https://doi.org/10.1145/3689236.3689272). [Online]. Available: <https://doi.org/10.1145/3689236.3689272>.
- [80] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu, *OSS-Fuzz*, Apr. 8, 2025. [Online]. Available: <https://github.com/google/oss-fuzz>.

- [81] Open Source Security Foundation (OpenSSF), *OSSF/fuzz-introspector*, Open Source Security Foundation (OpenSSF), Jun. 30, 2025. [Online]. Available: <https://github.com/ossf/fuzz-introspector>.
- [82] Python Software Foundation. “Venv — Creation of virtual environments,” Python documentation. (Jul. 17, 2025), [Online]. Available: <https://docs.python.org/3/library/venv.html>.
- [83] pip developers. “Pip documentation v25.1.1.” (2025), [Online]. Available: <https://pip.pypa.io/en/stable/>.
- [84] D. A. Wheeler. “Flawfinder Home Page,” Flawfinder. (), [Online]. Available: <https://dwheeler.com/flawfinder/>.
- [85] S. Zhao, Y. Yang, Z. Wang, Z. He, L. K. Qiu, and L. Qiu. “Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely.” arXiv: [2409.14924](https://arxiv.org/abs/2409.14924) [cs]. (Sep. 23, 2024), [Online]. Available: <http://arxiv.org/abs/2409.14924>, pre-published.
- [86] M. Chen, J. Tworek, H. Jun, *et al.* “Evaluating Large Language Models Trained on Code.” arXiv: [2107.03374](https://arxiv.org/abs/2107.03374) [cs]. (Jul. 14, 2021), [Online]. Available: <http://arxiv.org/abs/2107.03374>, pre-published.
- [87] A. Cedilnik, B. Hoffman, B. King, K. Martin, and A. Neundorff, *CMake - Upgrade Your Software Build System*, 2000. [Online]. Available: <https://cmake.org/>.
- [88] S. I. Feldman, “Make — a program for maintaining computer programs,” *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979, issn: 1097-024X. doi: [10.1002/spe.4380090402](https://doi.org/10.1002/spe.4380090402). [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402>.
- [89] T. He, *Sighingnow/libclang*, Jul. 3, 2025. [Online]. Available: <https://github.com/sighingnow/libclang>.
- [90] OpenAI Docs. “Text-embedding-3-small - OpenAI API.” (2025), [Online]. Available: <https://platform.openai.com>.
- [91] M. Douze, A. Guzhva, C. Deng, *et al.* “The Faiss library.” arXiv: [2401.08281](https://arxiv.org/abs/2401.08281) [cs]. (Feb. 11, 2025), [Online]. Available: <http://arxiv.org/abs/2401.08281>, pre-published.
- [92] O. Khattab, A. Singhvi, P. Maheshwari, *et al.* “DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines.” arXiv: [2310.03714](https://arxiv.org/abs/2310.03714) [cs]. (Oct. 5, 2023), [Online]. Available: <https://arxiv.org/abs/2310.03714>, pre-published.
- [93] Stanford NLP Team. “Signatures - DSPy Documentation.” (2025), [Online]. Available: <https://dsp.py.ai/learn/programming/signatures/>.
- [94] Stanford NLP Team. “ReAct - DSPy Documentation.” (2025), [Online]. Available: <https://dsp.py.ai/api/modules/ReAct/>.
- [95] H. Chase, *LangChain*, Oct. 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>.
- [96] J. Liu, *LlamaIndex*, Nov. 2022. doi: [10.5281/zenodo.1234](https://doi.org/10.5281/zenodo.1234). [Online]. Available: https://github.com/jerryjliu/llama_index.
- [97] F. Both. “Why we no longer use LangChain for building our AI agents.” (2024), [Online]. Available: <https://octomind.dev/blog/why-we-no-longer-use-langchain-for-building-our-ai-agents>.
- [98] M. Woolf. “The Problem With LangChain.” (Jul. 14, 2023), [Online]. Available: <https://minimaxir.com/2023/07/langchain-problem/>.
- [99] Woyera. “6 Reasons why Langchain Sucks,” Medium. (Sep. 8, 2023), [Online]. Available: <https://medium.com/@woyera/6-reasons-why-langchain-sucks-b6c99c98efbe>.
- [100] Astral, *Astral-sh/uv*, Astral, Jul. 18, 2025. [Online]. Available: <https://github.com/astral-sh/uv>.

- [101] Astral, *Astral-sh/ruff*, Astral, Jul. 18, 2025. [Online]. Available: <https://github.com/astral-sh/ruff>.
- [102] A. Cortesi, M. Hils, and T. Kriechbaumer, *Mitmproxy/pdoc*, mitmproxy, Jul. 18, 2025. [Online]. Available: <https://github.com/mitmproxy/pdoc>.
- [103] PyTest Dev Team, *Pytest-dev/pytest*, pytest-dev, Jul. 18, 2025. [Online]. Available: <https://github.com/pytest-dev/pytest>.
- [104] Python Software Foundation, *Python/mypy*, Python, Jul. 18, 2025. [Online]. Available: <https://github.com/python/mypy>.
- [105] Clibs Project. “Clib Packages,” GitHub. (2025), [Online]. Available: <https://github.com/clibs/club/wiki/Packages>.
- [106] Clibs Project, *Clibs/club*, clibs, Jul. 1, 2025. [Online]. Available: <https://github.com/clibs/club>.
- [107] OpenAI Docs. “GPT-4.1 mini - Open AI API.” (2025), [Online]. Available: <https://platform.openai.com>.
- [108] GitHub Docs. “About GitHub-hosted runners,” GitHub Docs. (2025), [Online]. Available: <https://docs-internal.github.com/en/actions/concepts/runners/about-github-hosted-runners>.
- [109] GitHub Docs. “Choosing the runner for a job,” GitHub Docs. (2025), [Online]. Available: <https://docs-internal.github.com/en/actions/how-tos/writing-workflows/choosing-where-your-workflow-runs/choosing-the-runner-for-a-job>.
- [110] O. I. Franksen, “Babbage and cryptography. Or, the mystery of Admiral Beaufort’s cipher,” *Mathematics and Computers in Simulation*, vol. 35, no. 4, pp. 327–367, 1993. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/037847549390063Z>.
- [111] F. Bacon, *Of the Proficiency and Advancement of Learning... Edited by the Rev. GW Kitchin*. Bell & Daldy, 1861.
- [112] T. Preston-Werner. “Semantic Versioning 2.0.0,” Semantic Versioning. (), [Online]. Available: <https://semver.org/>.
- [113] D. Giannone. “Demystifying AI Agents: ReAct-Style Agents vs Agentic Workflows,” Medium. (Feb. 9, 2025), [Online]. Available: <https://medium.com/@DanGiannone/demystifying-ai-agents-react-style-agents-vs-agentic-workflows-cedca7e26471>.
- [114] OpenAI Docs. “Model optimization - OpenAI API.” (2025), [Online]. Available: <https://platform.openai.com>.
- [115] S. Kim and S.-y. Lee, “Performance Comparison of Prompt Engineering and Fine-Tuning Approaches for Fuzz Driver Generation Using Large Language Models,” in *Innovative Mobile and Internet Services in Ubiquitous Computing*, L. Barolli, H.-C. Chen, and K. Yim, Eds., Cham: Springer Nature Switzerland, 2025, pp. 111–120, ISBN: 978-3-031-96093-2. DOI: [10.1007/978-3-031-96093-2_12](https://doi.org/10.1007/978-3-031-96093-2_12).
- [116] Z. Li, S. Dutta, and M. Naik. “IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities.” arXiv: [2405.17238](https://arxiv.org/abs/2405.17238) [cs]. (Apr. 6, 2025), [Online]. Available: <http://arxiv.org/abs/2405.17238>, pre-published.
- [117] D. Wang, G. Zhou, L. Chen, D. Li, and Y. Miao. “ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model.” arXiv: [2409.00922](https://arxiv.org/abs/2409.00922) [cs]. (Sep. 1, 2024), [Online]. Available: <http://arxiv.org/abs/2409.00922>, pre-published.

- [118] H. Green and T. Avgerinos, “GraphFuzz: Library API fuzzing with lifetime-aware dataflow graphs,” in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 1070–1081. doi: [10.1145/3510003.3510228](https://doi.org/10.1145/3510003.3510228). [Online]. Available: <https://dl.acm.org/doi/10.1145/3510003.3510228>.
- [119] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” presented at the USENIX Symposium on Operating Systems Design and Implementation, Dec. 8, 2008. [Online]. Available: <https://www.semanticscholar.org/paper/KLEE%3A-Unassisted-and-Automatic-Generation-of-Tests-Cadar-Dunbar/0b93657965e506dfbd56fbc1c1d4b9666b1d01c8>.
- [120] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang. “IntelliGen: Automatic Driver Synthesis for FuzzTesting.” arXiv: [2103.00862](https://arxiv.org/abs/2103.00862) [cs]. (Mar. 1, 2021), [Online]. Available: <http://arxiv.org/abs/2103.00862>, pre-published.
- [121] H. Xu, W. Ma, T. Zhou, *et al.* “CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph.” arXiv: [2411.11532](https://arxiv.org/abs/2411.11532) [cs]. (Dec. 20, 2024), [Online]. Available: <http://arxiv.org/abs/2411.11532>, pre-published.
- [122] N. Sasirekha, A. Edwin Robert, and M. Hemalatha, “Program Slicing Techniques and its Applications,” *International Journal of Software Engineering & Applications*, vol. 2, no. 3, pp. 50–64, Jul. 31, 2011, ISSN: 09762221. doi: [10.5121/ijsea.2011.2304](https://doi.org/10.5121/ijsea.2011.2304). [Online]. Available: <http://www.airccse.org/journal/ijsea/papers/0711ijsea04.pdf>.
- [123] OSS-Fuzz. “OSS-Fuzz Documentation,” OSS-Fuzz. (2025), [Online]. Available: <https://google.github.io/oss-fuzz/>.
- [124] Google, *Google/clusterfuzz*, Google, Apr. 9, 2025. [Online]. Available: <https://github.com/google/clusterfuzz>.
- [125] Google, *Google/fuzztest*, Google, Jul. 10, 2025. [Online]. Available: <https://github.com/google/fuzztest>.
- [126] Google, *Google/honggfuzz*, Google, Jul. 10, 2025. [Online]. Available: <https://github.com/google/honggfuzz>.
- [127] D. Liu, J. Metzman, O. Chang, and G. O. S. S. Team. “AI-Powered Fuzzing: Breaking the Bug Hunting Barrier,” Google Online Security Blog. (Aug. 16, 2023), [Online]. Available: <https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html>.
- [128] L. Thomason, *Leethomason/tinymxml2*, Jul. 10, 2025. [Online]. Available: <https://github.com/leethomason/tinymxml2>.
- [129] OSS-Fuzz Maintainers. “Introducing LLM-based harness synthesis for unfuzzed projects,” OSS-Fuzz blog. (May 27, 2024), [Online]. Available: <https://blog.oss-fuzz.com/posts/introducing-llm-based-harness-synthesis-for-unfuzzed-projects/>.
- [130] E. Martin, *Ninja-build/ninja*, ninja-build, Jul. 14, 2025. [Online]. Available: <https://github.com/ninja-build/ninja>.
- [131] J. Pakkanen, *Mesonbuild/meson*, The Meson Build System, Jul. 14, 2025. [Online]. Available: <https://github.com/mesonbuild/meson>.
- [132] Google, *Google/atheris*, Google, Apr. 9, 2025. [Online]. Available: <https://github.com/google/atheris>.

A. Abandoned Techniques

During its development, OverHAuL went through several iterations. A number of approaches were implemented and evaluated, with some being replaced for better alternatives. These are:

1. One-shot harness generation

Before the iterative feedback loop (Section 3.3.1) was implemented, OverHAuL attempted to operate in a straightforward pipeline, with just a “generator” agent being tasked to generate the harness. This meant that at either the compilation step or evaluation step, any failure resulted in the execution being terminated. This approach put too much responsibility in the response of a single LLM query, with results more often than not being unsatisfactory.

2. Chain-of-Thought LLM instances

The current implementation of ReAct agents has effectively supplanted the less effective Chain of Thought (COT) LLM modules [50]. This shift underscores a critical realization in the harness generation process: the primary challenge lies not in the creation of the harness itself, but rather in the necessity for real-time feedback during execution. This is the reason why first employing COT prompting offered limited observed improvements.

COT techniques are particularly advantageous when the task assigned to the LLM demands a more reflective, in-depth analysis. However, when it comes to tasks such as knowledge extraction from a codebase oracle and taking live feedback from the environment into consideration, ReAct agents demonstrate greater efficiency and effectiveness.

3. Source code concatenation

Initially, there was no implementation of a codebase oracle. Instead, the LLM agents operated with a Python string that contained a concatenation of all the collected source code. While this method proved effective for smaller and simpler projects, it encountered significant limitations when applied to more complex codebases. The primary challenge was the excessive consumption of the LLM’s context window, which hindered its ability to process and analyze larger codebases effectively. As a result, this approach became increasingly unsustainable as project complexity grew, underscoring the need for a more robust solution.

4. `{index, read}_tool` usage for ReAct agents

The predecessor of the oracle comprised a dual-system approach for code exploration, integrating the `index_tool` and the `read_tool`. The `index_tool` offered the LLM agent a

structured JSON object that delineated the project’s architecture, including all relevant file paths. On the other hand, the `read_tool` required a file path as input and returned the file’s content, albeit truncated to a maximum of 4000 characters. While this methodology presented an improvement in scalability over earlier systems, several limitations persisted.

Firstly, the LLM was constrained to searching through the codebase strictly in file-specific terms, which limited its efficacy in understanding the broader context of code relationships. Furthermore, the imposed character limit on the `read_tool` meant that certain portions of the codebase remained inaccessible, impeding the agent’s analytical capabilities. Even if this character limit were to be lifted, the resultant output would still occupy a significant portion of the context window, particularly in larger and more intricate projects. As such, while this approach offered advancements in code exploration, it still fell short.

B. Sample Generated Harnesses

This appendix presents a selection of harnesses generated by OverHAuL, originating from the benchmark run [16351915455](#). The examples included herein demonstrate a notable diversity, ranging from concise and straightforward harnesses to those that are more elaborate and complex. The crash inputs are provided in the form of their corresponding xxd hexadecimal representations.

B.1. `clibs/buffer`

Generated harness:

```
1  #include "buffer.h"
2  #include <stdint.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6
7  /*
8   * LLVMFuzzerTestOneInput harness to fuzz the function buffer_appendf.
9   *
10  * The fuzzing input data is treated as a format string argument to
11  * buffer_appendf. This targets potential format string vulnerabilities and
12  * buffer resizing issues within buffer_appendf as flagged by static analysis.
13  *
14  * We create a fresh buffer_t instance with buffer_new(), then call
15  * buffer_appendf() with the fuzz input as the format string. We do not supply
16  * additional variadic arguments as the input is a single buffer.
17  *
18  * After the call, we free the buffer_t instance to avoid memory leaks.
19  */
20  int
21  LLVMFuzzerTestOneInput (const uint8_t *data, size_t size)
22  {
23      if (size == 0)
24      {
```

```

25     // No data to test; skip
26     return 0;
27 }
28
29 // Allocate a writable null-terminated string for the format input
30 // We add one byte for the null terminator
31 char *fmt = (char *)malloc (size + 1);
32 if (!fmt)
33     return 0; // Allocation failed, skip
34
35 // Copy input and null terminate
36 memcpy (fmt, data, size);
37 fmt[size] = '\0';
38
39 // Create a new buffer instance
40 buffer_t *buf = buffer_new ();
41 if (!buf)
42 {
43     free (fmt);
44     return 0;
45 }
46
47 // Call buffer_appendf with user-controlled format string
48 // No additional arguments are supplied
49 (void)buffer_appendf (buf, fmt);
50
51 // Cleanup
52 buffer_free (buf);
53 free (fmt);
54
55 return 0;
56 }

```

Body of function that the harness targets:

```

1 int buffer_appendf(buffer_t *self, const char *format, ...) {
2     va_list ap;
3     va_list tmpa;
4     char *dst = NULL;
5     int length = 0;
6     int required = 0;
7     int bytes = 0;
8

```

```

9     va_start(ap, format);
10
11     length = buffer_length(self);
12
13     // First, we compute how many bytes are needed
14     // for the formatted string and allocate that
15     // much more space in the buffer.
16     va_copy(tmpa, ap);
17     required = vsnprintf(NULL, 0, format, tmpa);
18     va_end(tmpa);
19     if (-1 == buffer_resize(self, length + required)) {
20         va_end(ap);
21         return -1;
22     }
23
24     // Next format the string into the space that we
25     // have made room for.
26     dst = self->data + length;
27     bytes = vsnprintf(dst, 1 + required, format, ap);
28     va_end(ap);
29
30     return bytes < 0
31         ? -1
32         : 0;
33 }

```

Crash input which triggers an exception in `buffer_appendf()`>`vsnprintf()`:

```

00000000: 1b2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d  .-----
00000010: 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d  -----
00000020: 2d2d 2d2d 2d2d 2d2d 2d2d 2d25 2d2d 2d2d  -----%----
00000030: 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d      -----

```

B.2. willem/cbuffer

Generated harness:

```

1  #include "cbuffer.h"
2  #include <stddef.h>
3  #include <stdint.h>
4  #include <stdlib.h>

```

```

5  #include <string.h>
6
7  // New improved fuzzer harness for cbuf_offer with multiple writes and boundary
8  // tests
9  int
10 LLVMFuzzerTestOneInput (const uint8_t *data, size_t size)
11 {
12     if (size == 0)
13     {
14         // allow zero-size input to test zero-length write
15         cbuf_t *cbuf = cbuf_new (12);
16         if (cbuf == NULL)
17             return 0;
18         // calling offer with zero length should be tested
19         (void)cbuf_offer (cbuf, NULL, 0);
20         cbuf_free (cbuf);
21         return 0;
22     }
23
24     cbuf_t *cbuf = cbuf_new (12);
25     if (cbuf == NULL)
26         return 0;
27
28     // Split input roughly into two parts; first fills the buffer partially,
29     // second part is used for fuzzing cbuf_offer calls
30     size_t split = size / 2;
31     const uint8_t *prefix_data = data;
32     size_t prefix_size = split;
33     const uint8_t *main_data = data + split;
34     size_t main_size = size - split;
35
36     // Initially fill the buffer partially with prefix_data to simulate used
37     // space
38     if (prefix_size > 0)
39     {
40         int space = cbuf_unusedspace (cbuf);
41         int to_write
42             = prefix_size < (size_t)space ? (int)prefix_size : space - 1;
43         if (to_write > 0)
44         {
45             (void)cbuf_offer (cbuf, prefix_data, to_write);
46         }
47     }

```

```

48
49 // Now fuzz cbuf_offer with main_data
50 // Derive write size from first byte of main_data if available, else zero.
51 int write_size = 0;
52 if (main_size > 0)
53 {
54     write_size = main_data[0];
55     // Allow write size to be zero (edge case) and up to larger than buffer
56     // size to test rejection path Normalize write_size to a range: 0 to 2 *
57     // cbuf->size to test boundary and overflow cases clearly
58     int max_test_size = (int)(cbuf->size * 2);
59     write_size
60         = (write_size
61             % (max_test_size + 1)); // allows 0 to max_test_size inclusive
62 }
63
64 // Pointer to data for writing is after first byte in main_data if exists
65 const uint8_t *write_data = main_data + 1;
66 size_t write_data_len = (main_size > 0) ? main_size - 1 : 0;
67
68 // Clamp write_size to write_data_len but allow write_size > write_data_len
69 // to simulate out of bounds sizes by assigning write_data_len as is -- note
70 // cbuf_offer will only read up to write_size bytes anyway but fuzzing with
71 // invalid sizes tests boundary conditions.
72
73 if ((size_t)write_size > write_data_len)
74 {
75     // We keep write_size as is for boundary fuzzing (even if it overflows
76     // input), but avoid buffer overflow by passing smaller data pointer as
77     // NULL if no data available
78     if (write_data_len == 0)
79     {
80         write_data = NULL;
81     }
82 }
83
84 // Call cbuf_offer with fuzz parameters (including zero and oversized writes)
85 (void)cbuf_offer (cbuf, write_data, write_size);
86
87 // Additionally try a zero length call explicitly to test that edge case
88 (void)cbuf_offer (cbuf, NULL, 0);
89
90 // Optionally perform a few repeated calls with no data to simulate further

```



```

91 // state changes
92 (void)cbuf_offer (cbuf, NULL, 0);
93 (void)cbuf_offer (cbuf, NULL, 0);
94
95 cbuf_free (cbuf);
96 return 0;
97 }

```

Body of function that the harness targets:

```

1 int cbuf_offer(cbuf_t *me, const unsigned char *data, const int size)
2 {
3     /* prevent buffer from getting completely full or over committed */
4     if (cbuf_unusedspace(me) ≤ size)
5         return 0;
6
7     int written = cbuf_unusedspace(me);
8     written = size < written ? size : written;
9     memcpy(me→data + me→tail, data, written);
10    me→tail += written;
11    if (me→size < me→tail)
12        me→tail %= me→size;
13    return written;
14 }

```

Crash input which triggers an exception in cbuf_offer()>memcpy():

```
00000000: 0a
```

B.3. dvhar/dateparse

Generated harness:

```

1 #include <stddef.h>
2 #include <stdint.h>
3 #include "dateparse.h"
4
5 // LibFuzzer entry point
6 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
7     if (size == 0) return 0;
8

```

```

9     date_t parsed_date = 0;
10     int offset = 0;
11
12     // Pass the input data directly without copying
13     (void)dateparse((const char *)data, &parsed_date, &offset, (int)size);
14
15     return 0;
16 }

```

Body of function that the harness targets:

```

1 int dateparse(const char* datestr, date_t* t, int *offset, int stringlen){
2     struct parser p;
3     *t = 0;
4     if (!stringlen)
5         stringlen = strlen(datestr);
6     if (parseTime(datestr, &p, stringlen))
7         return -1;
8     return parse(&p, t, offset);
9 }

```

Crash input that causes a heap buffer overflow in `dateparse()`>`parseTime()`>`setMonth()`>`strncpy()`.

00000000: 6632 2066

f2 f

B.4. h2non/semver.c

Generated harness:

```

1 #include "semver.h"
2 #include <stdint.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 // Enhanced fuzzer harness for semver_parse including calls to semver_numeric
7 // and semver_render to help increase coverage and trigger more bugs faster.
8 // Dynamically allocates input buffer with no size limit other than system
9 // memory. Frees allocated semver_t string members after use.
10 int
11 LLVMFuzzerTestOneInput (const uint8_t *data, size_t size)
12 {

```

```

13 // Allocate buffer dynamically to handle any input size.
14 char *input_str = (char *)malloc (size + 1);
15 if (input_str == NULL)
16 {
17     return 0;
18 }
19 memcpy (input_str, data, size);
20 input_str[size] = '\0';
21
22 semver_t ver;
23 memset (&ver, 0, sizeof (ver));
24
25 // Parse the semver string.
26 (void)semver_parse (input_str, &ver);
27
28 // Call semver_numeric to exercise more code paths.
29 (void)semver_numeric (&ver);
30
31 // Call semver_render with buffer large enough for rendered output.
32 // Typical semantic version strings are short, but use a buffer of size
33 // size+20 to be safe.
34 char *render_buf = (char *)malloc (size + 20);
35 if (render_buf != NULL)
36 {
37     memset (render_buf, 0, size + 20);
38     semver_render (&ver, render_buf);
39     free (render_buf);
40 }
41
42 // Free dynamically allocated members inside semver_t.
43 if (ver.prerelease)
44 {
45     free (ver.prerelease);
46     ver.prerelease = NULL;
47 }
48 if (ver.metadata)
49 {
50     free (ver.metadata);
51     ver.metadata = NULL;
52 }
53
54 free (input_str);
55 return 0;

```

```
56 }
```

Bodies of functions that the harness targets:

```
1  /**
2   * Parses a string as semver expression.
3   *
4   * Returns:
5   *
6   * `0` - Parsed successfully
7   * `-1` - In case of error
8   */
9
10 int
11 semver_parse (const char *str, semver_t *ver)
12 {
13     int valid, res;
14     size_t len;
15     char *buf;
16     valid = semver_is_valid (str);
17     if (!valid)
18         return -1;
19
20     len = strlen (str);
21     buf = (char *)calloc (len + 1, sizeof (*buf));
22     if (buf == NULL)
23         return -1;
24     strcpy (buf, str);
25
26     ver->metadata = parse_slice (buf, MT_DELIMITER[0]);
27     ver->prerelease = parse_slice (buf, PR_DELIMITER[0]);
28
29     res = semver_parse_version (buf, ver);
30     free (buf);
31     #if DEBUG > 0
32         printf ("[debug] semver.c %s = %d.%d.%d, %s %s\n", str, ver->major,
33             ver->minor, ver->patch, ver->prerelease, ver->metadata);
34     #endif
35     return res;
36 }
37
38 // ...
39
```

```

40  /**
41   * Render a given semver as string
42   */
43
44  void
45  semver_render (semver_t *x, char *dest)
46  {
47      concat_num (dest, x->major, NULL);
48      concat_num (dest, x->minor, DELIMITER);
49      concat_num (dest, x->patch, DELIMITER);
50      if (x->prerelease)
51          concat_char (dest, x->prerelease, PR_DELIMITER);
52      if (x->metadata)
53          concat_char (dest, x->metadata, MT_DELIMITER);
54  }

```

Crash input that causes a stack buffer overflow in `semver_render()`>`concat_char()`>`sprintf()`:

```

00000000: 392d 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 9-+++++
00000010: 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b +++++
00000020: 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b +++++
00000030: 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b +++++
00000040: 2b2b 2b2b 2b2b 2b46 4c                +++++FL

```

C. DSPy Custom Signatures

```
1  class GenerateHarness(dspy.Signature):
2      """
3      You are an experienced C/C++ security testing engineer. You must write a
4      libFuzzer-compatible `int LLVMFuzzerTestOneInput(const uint8_t *data, size_t
5      size)` harness for a function of the given C project. Your goal is for the
6      harness to be ready for compilation and for it to find successfully a bug in
7      the function-under-test. Write verbose (within reason) and helpful comments
8      on each step/decision you take/make, especially if you use "weird" constants
9      or values that have something to do with the project.
10
11     You have access to a rag_tool, which contains a vector store of
12     function-level chunks of the project. Use it to write better harnesses. Keep
13     in mind that it can only reply with function chunks, do not ask it to
14     combine stuff.
15
16     The rag_tool does not store any information on which lines the functions
17     are. So do not ask questions based on lines.
18
19     Make sure that you only fuzz an existing function. You will know that a
20     functions exists when the rag_tool returns to you its signature and body.
21     """
22
23     static: str = dspy.InputField(
24         desc=""" Output of static analysis tools for the project. If you find it
25         helpful, write your harness so that it leverages some of the potential
26         vulnerabilities described below. """
27     )
28     new_harness: str = dspy.OutputField(
29         desc=""" C code for a libFuzzer-compatible harness. Output only the C
30         code, **DO NOT format it in a markdown code cell with backticks**, so
31         that it will be ready for compilation.
32
33         <important>
34
35         Add **all** the necessary includes, either project-specific or standard
```

```

36     libraries like <string.h>, <stdint.h> and <stdlib.h>. Also include any
37     header files that are part of the project and are probably useful. Most
38     projects have a header file with the same name as the project at the
39     root.
40
41     **The function to be fuzzed absolutely must be part of the source
42     code**, do not write a harness for your own functions or speculate about
43     existing ones. You must be sure that the function that is fuzzed exists
44     in the source code. Use your rag tool to query the source code.
45
46     Do not try to fuzz functions of the project that are static, since they
47     are only visible in the file that they were declared. Choose other
48     user-facing functions instead.
49
50     </important>
51
52     **Do not truncate the input to a smaller size than the original**,
53     e.g. for avoiding large stack usage or to avoid excessive buffers. Opt
54     to using the heap when possible to increase the chance of exposing
55     memory errors of the library, e.g. mmap instead of declaring
56     buf[1024]. Any edge cases should be handled by the library itself, not
57     the harness. On the other hand, do not write code that will most
58     probably crash irregardless of the library under test. The point is for
59     a function of the library under test to crash, not the harness
60     itself. Use and take advantage of any custom structs that the library
61     declares.
62
63     Do not copy function declarations inside the harness. The harness will
64     be compiled in the root directory of the project. """
65 )
66
67
68 class FixHarness(dspy.Signature):
69     """
70     You are an experienced C/C++ security testing engineer. Given a
71     libFuzzer-compatible harness that fails to compile and its compilation
72     errors, rewrite it so that it compiles successfully. Analyze the compilation
73     errors carefully and find the root causes. Add any missing #includes like
74     <string.h>, <stdint.h> and <stdlib.h> and #define required macros or
75     constants in the fuzz target. If needed, re-declare functions or struct
76     types. Add verbose comments to explain what you changed and why.
77     """
78

```

```

79     old_harness: str = dspy.InputField(desc="The harness to be fixed.")
80     error: str = dspy.InputField(desc="The compilaton error of the harness.")
81     new_harness: str = dspy.OutputField(
82         desc="""The newly created harness with the necessary modifications for
83         correct compilation."""
84     )
85
86
87     class ImproveHarness(dspy.Signature):
88         f"""
89         You are an experienced C/C++ security testing engineer. Given a
90         libFuzzer-compatible harness that does not find any bug/does not crash (even
91         after running for {Config.EXECUTION_TIMEOUT} seconds) or has memory leaks
92         (generates leak files), you are called to rewrite it and improve it so that
93         a bug can be found more easily and/or memory is managed correctly. Determine
94         the information you need to write an effective fuzz target and understand
95         constraints and edge cases in the source code to do it more
96         effectively. Reply only with the source code --- without backticks. Add
97         verbose comments to explain what you changed and why.
98         """
99
100     old_harness: str = dspy.InputField(
101         desc="The harness to be improved so it can find a bug more quickly."
102     )
103     output: str = dspy.InputField(desc="The output of the harness' execution.")
104     new_harness: str = dspy.OutputField(
105         desc="""The newly created harness with the necessary modifications for
106         quicker bug-finding. If the provided harness has unnecessary input
107         limitations regarding size or format etc., remove them."""
108     )

```