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

Lorem ipsum odor amet, consectetuer adipiscing elit. Habitasse congue tempus erat rhoncus sapien interdum dolor nec. Posuere habitant metus tellus erat eu. Risus ultricies eu rhoncus, conubia euismod convallis commodo per. Nam tellus quisque maximus duis eleifend; arcu aptent. Nisi rutrum primis luctus tortor tempor maecenas. Donec curae cras dolor; malesuada ultricies scelerisque. Molestie class tincidunt quis gravida ut proin. Consequat lacinia arcu justo leo maecenas nunc neque ex. Platea eros ullamcorper nullam rutrum facilisis.

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 Phaedon, 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

List of Figures

List of Listings

List of Tables

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 began 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 outlining the foundational concepts necessary to understand its context (Chapter 2) and progresses to a thorough survey of existing research in the field of automated fuzzing (Chapter 3). We illustrate that the majority of contemporary fuzzing systems either depend on pre-existing harnesses or utilize client code, frequently placing the burden of validation and integration on the user. Next, we present the OverHAuL system, detailing its architecture and the innovative techniques that underpin its implementation, as well as their contributions to the advancement of automated harness generation (Chapter 4). Lastly, we compile a benchmark dataset consisting of ten open-source C projects and rigorously assess OverHAuL's performance (Chapter 5, 6).

1.2. Summary of Contributions

This thesis presents the following key contributions:

1. A comprehensive, up-to-date survey of leading LLM-driven automated fuzzing tools, detailing their respective strengths and limitations.
2. The introduction of OverHAuL, a novel framework that enables fully automated, end-to-end fuzzing harness generation using LLMs.
3. Empirical validation through benchmarking experiments, demonstrating that OverHAuL successfully generates effective fuzzing harnesses in 92.5% of tested cases.
4. 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.

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.

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.

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

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.

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*.

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-read[1] 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*—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 [29].

[1] https://xkcd.com/1354/

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++ [30]—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 [31] 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.

```c
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithData(Data, Size);
  return 0;
}
```

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

```cpp
// test_fuzzer.cpp
#include <stdint.h>
#include <stddef.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  if (size > 0 && data[0] == 'H')
    if (size > 1 && data[1] == 'I')
      if (size > 2 && data[2] == '!')
        __builtin_trap();
  return 0;
}
```

To compile and link such a harness with LibFuzzer, the Clang compiler—also part of the LLVM project [31]—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.

```bash
# Compile test_fuzzer.cc with AddressSanitizer and link against LibFuzzer.
clang++ -fsanitize=address,fuzzer test_fuzzer.cc
# Execute the fuzzer without any pre-existing seed corpus.
./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 [32]—an open-source CPU emulator facilitating fuzzing on diverse architectures—and Unicorn [33], a lightweight multi-platform CPU emulator. While AFL established itself as a foundational tool within the fuzzing community, its successor AFL++ [30] 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:

```
./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 underutilized 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.

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 [34], [35]. 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 [36]. 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 [37]. 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 [38]. 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 [39], further pushed the boundaries. Subsequent iterations, including GPT-2 [40], GPT-3 [41], and the most current GPT-4 [42], 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) [43]–[45]. The proliferation of LLMs has sparked an active discourse about their capabilities, applications, and implications in various fields.

2.2.1. Biggest GPTs

User-facing LLMs are generally categorized between closed-source and open-source models. Closed-source LLMs like ChatGPT, Claude, and Gemini [43], [46], [47] represent commercially developed systems often optimized for specific tasks without public access to their underlying weights. In contrast, open-source models[2], including the Llama series [45] and Deepseek [44], 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, 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 a specific task without any examples, while in few-shot prompting, the user provides a limited number of examples to guide the model's responses [41].

To enhance performance on more complex tasks, several advanced prompting techniques have emerged. One notable strategy is the *Chain of Thought* approach [48], 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 refined variant of this approach is the *Tree of Thoughts* technique [49], which

[2]The 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.

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) [50] 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 [51], 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 tools. 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, which leverage LLM capabilities to provide developers with coding suggestions, auto-completions, and even real-time debugging assistance [52], [53]. Such innovations have introduced a layer of interaction that enhances productivity and fosters a more intuitive coding experience. Simultaneously, certain LLMs are trained themselves with the code-generation task in mind [54]–[56].

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 harnessed to elevate the coding experience by supporting developers as they navigate complex programming tasks [57]. 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". 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. 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

While large language models (LLMs) demonstrate significant potential in enhancing the software development process, the challenges highlighted in Section 2.2.3 become even more pronounced and troublesome when these models are employed to generate fuzzing harnesses. The task of writing a fuzzing harness inherently demands an in-depth comprehension of both the library being tested and the intricate interactions expected among its various components. This level of understanding is often beyond the capabilities of LLMs, primarily due to their context window limitations, which restrict the amount of information they can effectively process and retain during code generation.

In addition to this issue, the risk of error-prone code produced by LLMs further complicates the fuzzing workflow. When a crash occurs during the fuzzing process, it becomes imperative for developers to ascertain that the root cause of the failure is not attributable to deficiencies or bugs within the harness itself. This additional layer of verification adds to the cognitive load placed upon developers, potentially detracting from their ability to focus on testing and improving the underlying software.

To enhance the reliability of LLM-generated harnesses in fuzzing contexts, it is essential that these generated artifacts undergo thorough evaluation and validation through programmatic means. This involves the implementation of systematic techniques that assess the accuracy and robustness of the generated code, ensuring that it aligns with the expected behavior of the components it is intended to interact with. This strategy can be conceptualized within the framework of Neurosymbolic AI (Section 2.3), which seeks to integrate the strengths of neural networks with symbolic reasoning capabilities. By marrying these two paradigms, it may be possible to improve the reliability and efficacy of LLMs in the creation of fuzzing harnesses, ultimately leading to a more seamless integration of automated testing methodologies into the software development lifecycle.

2.3. Neurosymbolic AI

Neurosymbolic AI (NeSy 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 [58], [59]. 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, NeSy AI aims to create cognitive models that are not only more accurate but also more robust in problem-solving contexts.

At its core, NeSy AI facilitates the development of AI systems that are capable of understanding and interpreting feedback in real-world scenarios [60]. This characteristic is particularly significant in the current landscape of artificial intelligence, where LLMs are predominant. In this context, NeSy AI is increasingly viewed as a critical solution to pressing issues related to explainability, attribution, and reliability in AI systems [61], [62]. 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, NeSy AI can significantly contribute to the broader landscape of trustworthy AI systems, allowing for more transparent and accountable decision-making processes [58], [61], [62].

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 3). Among these, OverHAuL constitutes a Neuro[Symbolic] tool, as classified by Henry Kautz's taxonomy [63], [64]. This means that the neural model—specifically the LLM—can leverage symbolic reasoning tools—in this case a source code explorer (Chapter 7)—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. 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 [37] and the 2020's boom of commercial GPTs [46], 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, in the name of automation [9]–[12], [65]–[70].

3.1. Previous Projects

3.1.1. KLEE

KLEE [71] 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 [31] bytecode representation of programs, allowing it to be applied to a wide range of C programs compiled to the LLVM intermediate representation.

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.

3.1.2. IRIS

IRIS [65] 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.

3.1.3. FUDGE

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 [72]. 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* [73], per FUDGE, based on its Abstract Syntax Tree (AST) using LLVM's Clang tool [31]. 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.

3.1.4. UTopia

UTopia [10] (stylized UTopia) is another open-source automatic harness generation framework. Aside from the library code, It operates solely on user-provided unit tests since, according to Jeong, Jang, Yi, *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.

3.1.5. FuzzGen

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 consuming and analyzing client code and tests that concern the library under test, 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 that uses it. 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^2DGs 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 A^2DG. This graph can be seen as a template for correct usage of the library. *3. Fuzzer generator.* Through the A^2DG, 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 of the A^2DG. In other words, while FUDGE fuzzes a single API call at a time, FuzzGen's result is a single harness that tries to fuzz the given library all at once through complex API usage.

3.1.6. IntelliGen

IntelliGen [74] is a system for automatically synthesizing fuzz drivers by statically identifying potentially vulnerable entry-point functions within C projects. Implemented using LLVM [31], 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 two main components: the *Entry Function Locator* and the *Fuzz Driver Synthesizer.* The Entry Function Locator analyzes the project's abstract syntax tree (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.

IntelliGen stands out by integrating static vulnerability estimation into the driver generation pipeline. Compared to prior tools like FuzzGen and FUDGE, it uses a more targeted, heuristic-based selection of functions, increasing the likelihood that fuzzing will exercise meaningful and vulnerable code paths.

3.1.7. CKGFuzzer

CKGFuzzer [75] is a fuzzing framework designed to automate the generation of effective fuzz drivers for C/C++ libraries by leveraging static analysis and large language models. 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 abstract syntax tree (AST), and second, performing inter-procedural program analysis. Through this process, CKGFuzzer extracts essential program elements such as data structures, function signatures, function 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 during this phase 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 at the file level. 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 (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 Common Weakness Enumeration (CWE) entries.

3.1.8. PromptFuzz

PromptFuzz [76] constitutes a framework for automatically generating fuzz drivers using LLMs, with a novel focus on *prompt mutation* to improve coverage. The system is implemented in Rust and targets C libraries, aiming to explore more of the API surface with each iteration.

The workflow begins with the random selection of API functions, extracted from header file declarations. These functions are used to construct initial prompts that instruct the LLM to generate a simple program utilizing the API. Each generated program is compiled, executed, and monitored for code coverage. Programs that fail to compile or violate runtime checks (e.g. sanitizers) are discarded.

A key innovation in PromptFuzz is *coverage-guided prompt mutation*. Instead of mutating generated code directly, PromptFuzz mutates the LLM prompts—selecting new combinations of API functions to target unexplored code paths. This process is guided by a *power scheduling* strategy that prioritizes underused or promising API functions based on feedback from previous runs.

Once an effective program is produced, it is transformed into a fuzz driver by replacing constants and arguments with variables derived from the fuzzer input. Multiple such drivers are embedded into a single harness, where the input determines which program variant to execute, typically via a case-switch construct.

Overall, PromptFuzz demonstrates that prompt-level mutation enables more effective exploration of complex APIs and achieves better coverage than direct code mutations, offering a compelling direction for LLM-based automated fuzzing systems.

3.1.9. OSS-Fuzz

OSS-Fuzz [72], [77] 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 [78] project. ClusterFuzz is the fuzzing infrastructure that OSS-Fuzz uses under the hood and depends on Google Cloud Platform services, although it can be hosted 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], [30], [79], [80]. C, C++, Rust, Go, Python and Java/JVM projects are supported.

3.1.10. OSS-Fuzz-Gen

OSS-Fuzz-Gen (OFG) [9], [81] is Google's current State-Of-The-Art (SOTA) project regarding automatic harness generation through LLMs. It's 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 [82]. 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 proved meaningful, with code coverage in fuzz campaigns increasing thanks to the new generated fuzz drivers. In the case of [83], line coverage went from 38% to 69% without any manual interventions [81].

In 2024, OSS-Fuzz-Gen introduced an experimental feature for generating harnesses in previously unfuzzed projects [84]. 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.

3.1.11. AutoGen

AutoGen [66] is a closed-source tool that generates new fuzzing harnesses, given only the library code and documentation. It works as following: 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 documentation. 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.

4. OverHAuL's Design

In this thesis we present *OverHAuL* (**H**arness **Au**tomation 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 three LLM ReAct agents [51]—each with its own responsibility and scope—and a vector store index reserving the given project's analyzed codebase. An overview of OverHAuL's process is presented in Figure 4.1. The objective of OverHAuL is to streamline the process of fuzz testing for C libraries. Given a link to a git repository [85] 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.

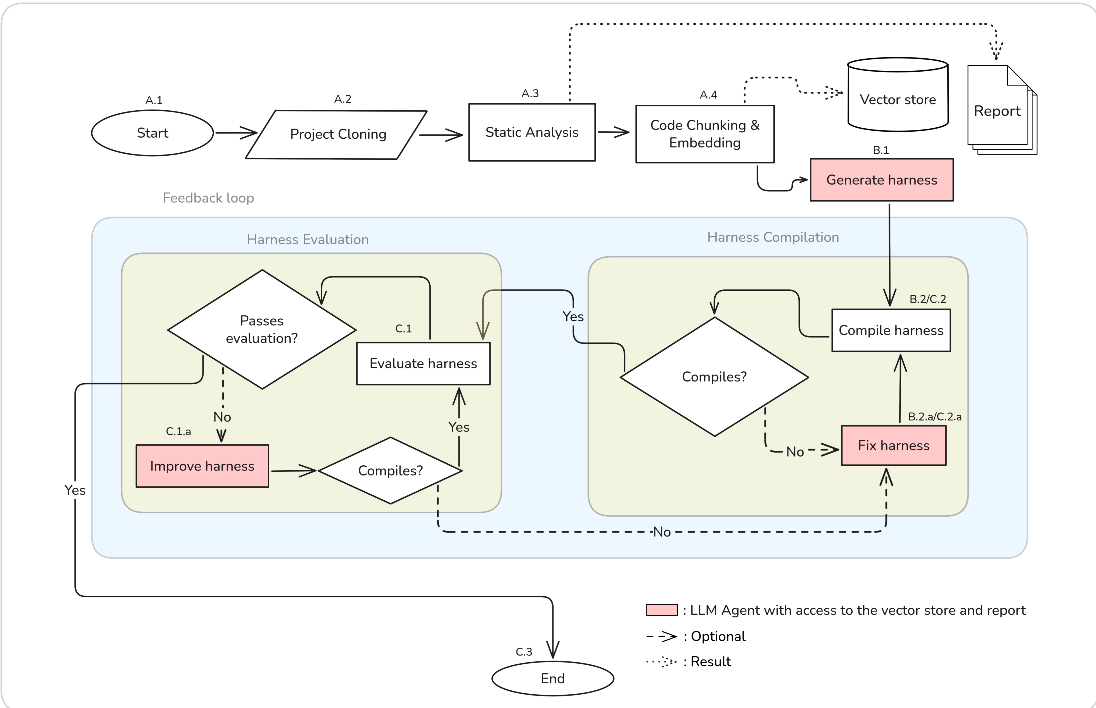

Figure 4.1.: Overview of OverHAuL's automatic harnessing process.

As commented in Section 4.4, 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 [66]. Also importantly, OverHAuL is decoupled from other fuzzing projects, thus lowering the barrier to entry for new projects [9], [72]. Lastly, the user isn't mandated to specify manually 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 targeted function.

OverHAuL utilizes autonomous ReAct agents [51] which inspect and analyze the project's source code. The latter is stored and interacted with as a set of text embeddings [86], 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.

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]

4.1. Architecture

OverHAuL can be compartmentalized in three stages: First, the project analysis stage (Section 4.1.1), the harness creation stage (Section 4.1.2) and the harness evaluation stage (Section 4.1.3).

4.1.1. Project Analysis

In the project analysis stage (steps A.1–A.4), the project to be fuzzed is ran through a static analysis tool 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 vector store containing embeddings of function-level code chunks, both of which are later available to the LLM agents.

The static analysis tool Flawfinder [87] is executed with the project directory as input and is responsible for the static analysis report. This report is considered a meaningful resource, since it provides the LLM agent with some starting points to explore, regarding the occurrences of potentially vulnerable functions and/or unsafe code practices.

The vector store is created in the following manner: The codebase 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. Afterwards, each function body is turned into a vector embedding through an

[1] I.e. "research code".

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 the LLMs.

4.1.2. 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 vector store. 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 4.6. 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.

4.1.3. 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.

4.2. Main 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 swarm of distinct agents and the employment of a “codebase oracle” for interacting with the given project’s source code.

4.2.1. 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 4.1 is followed. Such an approach allows for targeted improvements while maintaining oversight of resource allocation throughout the harness development cycle.

4.2.2. React Agents Swarm

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.

4.2.3. 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 [88]. 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 [89]. 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.

4.3. High-Level Algorithm

A pseudocode version of OverHAuL’s main function can be seen in Algorithm 4.1. It represents the workflow presented in Figure 4.1 and uses the techniques described in sections 4.1 and 4.2. It is important to emphasize that, within the context of this algorithm, the `HarnessAgents()` function serves as an interface that bridges the “generator”, “fixer” and “improver” LLM agents. The agent that is used upon each function call depends on the values of the function’s arguments. This results in the *harness* variable representing all generated, fixed or improved harnesses. This approach is adopted for making the abstract algorithm simpler and easier to understand.

Algorithm 4.1 OverHAuL

```
Require: repository
Ensure: harness, compilation_script, crash_input, execution_log
path ← RepoClone(repository)
report ← StaticAnalysis(path)
vector_store ← CreateOracle(path)
acceptable ← False
compiled ← False
error ← None
violation ← None
output ← None
for i = 1 to MAX_ITERATIONS do
    harness ← HarnessAgents(path, report, vector_store, error, violation, output)
    error, compiled ← BuildHarness(path, harness)
    if ¬compiled then
        continue                                              ▷ Fix harness
    end if
    output, accepted ← EvaluateHarness(path, harness)
    if ¬accepted then
        continue                                              ▷ Improve harness
    else
        acceptable ← True
        break
    end if
end for
return compiled ∧ acceptable
```

4.4. Differences

OverHAuL differs, in some way, with each of the aforementioned works in Chapter 3. Firstly, although KLEE and IRIS [65], [71] 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 by default, 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.

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 [72], 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 [78] 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 [77]: "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 <code> 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 [66]. 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 [90] 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.

4.5. 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 [91] and installing it inside the environment using Python's PIP package installer [92], like in Listing 4.1.

Listing 4.1 OverHAuL's installation process.

```
$ git clone https://github.com/kchousos/overhaul; cd overhaul
  ...
$ python3.10 -m venv .venv
$ source ./.venv/bin/activate
$ pip install .
  ...
$ overhaul --help
usage: overhaul [-h] [-c COMMIT] [-m MODEL] [-f FILES [FILES ...]] [-o OUTPUT_DIR] repo

Generate fuzzing harnesses for C/C++ projects

positional arguments:
  repo                  Link of a project's git repo, for which to generate a harness.

options:
  -h, --help            show this help message and exit
  -c COMMIT, --commit COMMIT
                        A specific commit of the project to check out
  -m MODEL, --model MODEL
                        LLM model to be used. Available: o3-mini, o3, gpt-4o,
                        gpt-4o-mini, gpt-4.1, gpt-4.1-mini, gpt-3.5-turbo, gpt-4
  -f FILES [FILES ...], --files FILES [FILES ...]
                        File patterns to include in analysis (e.g. *.c *.h)
  -o OUTPUT_DIR, --output-dir OUTPUT_DIR
                        Directory to clone the project into. Defaults to "output"
$
```

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, like so:

```
# cat .env
OPENAI_API_KEY=<API-key-here>
```

Or it can be exported in the shell environment:

```
$ export OPENAI_API_KEY=<API-key-here>
$ overhaul <repo-link>
```

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. A sample successful execution can is presented in Figure 4.2.

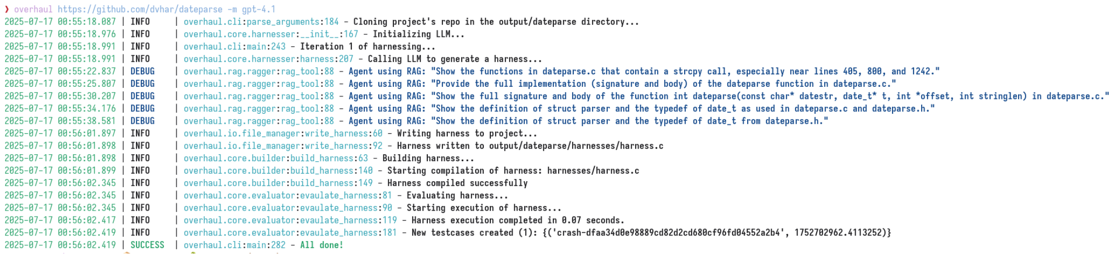

Figure 4.2.: A successful execution of OverHAuL, harnessing dvhar's dateparsing C library, using OpenAI's gpt-4.1 model. Debug statements are printed to showcase the interaction between the LLM agents and the codebase oracle (Section 4.2.3).

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, this 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 directory 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.

4.6. 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 [93], [94] is yet implemented. Such functionality would exponentially increase the complexity of the build step and is beyond the scope of this thesis.

5. 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?

5.1. Experimental Benchmark

To evaluate OverHAuL, a benchmarking script was implemented[1] and a corpus of ten open-source C libraries was assembled. This collection comprises of: Firstly, GitHub user dhvar's dateparse library, which is also used as a running example in OSS-Fuzz-Gen's [9] experimental from-scratch harnessing feature (Section 3.1.10). Secondly, nine other libraries chosen randomly[2] from the package catalog of Clib, a "package manager for the C programming language" [95], [96]. All libraries can be seen Table 5.1, along with their descriptions.

Table 5.1.: The benchmark project corpus. Each project name links to its corresponding GitHub repository. Each is followed by a short description and its GitHub stars count, as of July 18th, 2025.

Project	Description	Stars
dvhar/dateparse	A library that allows parsing dates without knowing the format in advance.	2
clibs/buffer	A string manipulation library.	204
jwerle/libbeaufort	A library implementation of the Beaufort cipher [97].	13

[1] https://github.com/kchousos/OverHAuL/blob/master/benchmarks/benchmark.sh

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

Project	Description	Stars
jwerle/libbacon	A library implementation of the Baconian cipher [98].	8
jwerle/chfreq.c	A library for computing the character frequency in a string.	5
jwerle/progress.c	A library for displaying progress bars in the terminal.	76
willemt/cbuffer	A circular buffer implementation.	261
willemt/torrent-reader	A torrent-file reader library.	6
orangeduck/mpc	A type-generic parser combinator library.	2,753
h2non/semver.c	A semantic version v2.0 parsing and rendering library [99].	190

5.1.1. Local Benchmarking

To run the benchmark locally, one would need to follow the installation instructions in Section 4.5 and then execute the benchmarking script, like so:

```
$ ./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.

6. Results

OverHAuL was evaluated through the experimental benchmark (Section 5.1) from 6th of June, 2025 to 18th of July, 2025, using OpenAI's gpt-4.1-mini model [100]. 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, and the result directory (described in Section 5.1.1) for each is available as a downloadable artifact in the corresponding GitHub Actions entry. In Figure 6.1, the results of these benchmark runs are showcased.

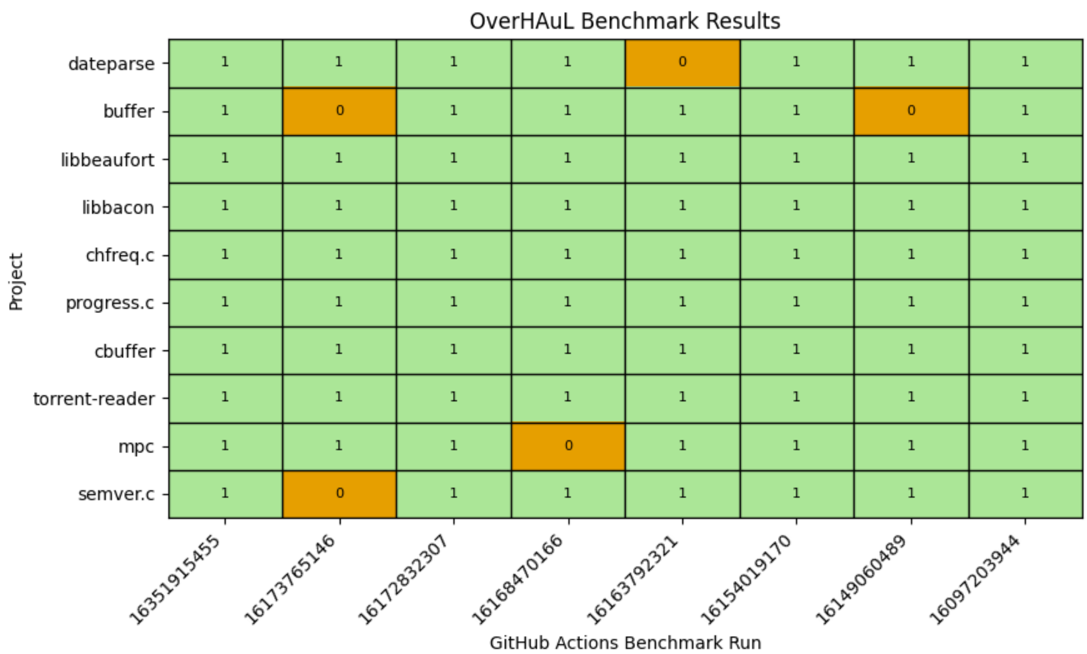

Figure 6.1.: The benchmark results for OverHAuL are illustrated with the y-axis depicting the ten-project corpus outlined in Section 5.1. The x-axis represents the various benchmark runs. Each label constitutes a unique hash identifier corresponding to a specific GitHub Actions workflow run, which can 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, an orange/0 block indicates that while a compilable harness was produced, no crash input was found within the five-minute execution period. Importantly, there are no red/-1 blocks, which would indicate cases where a compilable harness could not be generated.

6.1. Answers to RQs

We can deduce the following facts from Figure 6.1: Firstly, OverHAuL has a very high success rate in finding crash inputs for the given programs, specifically one of **92.5%**. Secondly, although according strictly to these results OverHAuL never generates an un-compilable harness, we can err on the side of caution and infer that OverHAuL not being able to generate a compilable harness is a rare occurrence. These conclusions can safely answer RQ1 with a resounding yes.

From sampling OverHAuL's generated harnesses, the answer to RQ2 remains unclear. Most of the time, the fuzz targets that are produced are understandable and similar to something a software engineer might program. Take for example Listing 6.1. Nonetheless, sometimes generated harnesses contain usage of inexplicable or arbitrary constants and peculiar control flow checks. This makes them harder to understand and quite possibly incorrect in many cases, thus diverging from seeming human-written. RQ2's answer remains an unclear "it depends", given the variance in OverHAuL's results.

In exploring the utilization of LLMs, two critical dimensions warrant examination: the selection of the LLM model itself and the prompting techniques employed. Within the context of model selection, all benchmark tests conducted on GitHub's infrastructure utilized OpenAI's gpt-4.1-mini. Additionally, local testing involved gpt-4.1, gpt-4o, gpt-4, and gpt-3.5-turbo. The results indicate that both gpt-4.1 and gpt-4.1-mini demonstrated comparable positive outcomes, while gpt-4o produced semi-average results. In contrast, gpt-4 and gpt-3.5-turbo exhibited significantly poorer performance, averaging approximately 2 out of 10 successfully harnessed projects per benchmark run. This underscores the considerable impact that the size and capabilities of the underlying LLM model have on OverHAuL's effectiveness. Consequently, gpt-4o emerges as a contemporary cut-off point in LLM development concerning OverHAuL's performance, suggesting that advancements in LLM technologies will likely enhance OverHAuL's capabilities rapidly.

Regarding the prompting techniques adopted, ReAct prompting has yielded the most favorable outcomes in the current version of OverHAuL [51]. As detailed in Appendix A, both zero-shot prompting and Chain-of-Thought (COT) prompting [48] produced similarly unsatisfactory results. A primary challenge in automatic harness generation is ensuring that the generated harness aligns with real-world conditions, particularly in terms of compilation success and effective runtime behavior. This alignment can only be validated through LLM–environment interaction, i.e. within agentic workflows [101]. Furthermore, the superior results associated with ReAct prompting can be attributed to its inherent capacity for more sophisticated code exploration.

In summary, the response to RQ3 comes to be that ongoing advancements in LLM models will enable systems like OverHAuL to generate increasingly effective outcomes. Additionally, architectures that incorporate agentic modules capable of environment assessment and feedback gathering will surpass more traditional applications of LLMs, particularly in the domain of automatic fuzzing.

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.

Listing 6.1 Sample harness for dateparse, generated by OverHAuL.

```c
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include "dateparse.h"

// No artificial size limit to allow fuzzer to explore full input size for boundaries

int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Allocate buffer with extra byte for null termination
    char *input_str = (char *)malloc(size + 1);
    if (!input_str) {
        return 0;
    }

    memcpy(input_str, data, size);
    input_str[size] = '\0';

    date_t parsed_date = 0;
    int offset = 0;

    // Array of string lengths targeting boundary conditions (including 0 = internal strlen)
    size_t test_lens[] = {0, size, size > 0 ? size - 1 : 0, 12, 13, 14};

    for (size_t i = 0; i < sizeof(test_lens) / sizeof(test_lens[0]); i++) {
        size_t len = test_lens[i];
        if (len ≤ size) {
            dateparse(input_str, &parsed_date, &offset, (int)len);
        }
    }

    free(input_str);
    return 0;
}
```

7. Implementation

In creating the codebase oracle, we employ the “libclang” Python package [102] to slice functions based on the AST capability provided by Clang. As detailed in Section 4.2.3, 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 [103] and stored in a FAISS vector store index [104]. 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 [90]. DSPy offers built-in modules and abstractions that facilitate the composition of LLMs and prompting techniques, such as Chain of Thought and ReAct (Listing 7.1). Each agent within OverHAuL is an instance of DSPy’s ReAct module [105], accompanied by a custom Signature [106]—displayed in Appendix B. DSPy was selected over other contemporary LLM libraries, such as LangChain and Llamaindex [107], [108], because of its user-friendliness, logical abstractions, and efficient development process—qualities that are often lacking in these alternative libraries [109]–[111].

Listing 7.1 Sample DSPy program.

```python
import dspy
lm = dspy.LM('openai/gpt-4o-mini', api_key='YOUR_OPENAI_API_KEY')
dspy.configure(lm=lm)

math = dspy.ChainOfThought("question -> answer: float")
math(question="Two dice are tossed. What is the probability that the sum equals two?")
```

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

7.1. 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 documentation[1] [112]–[116].

7.2. 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 5.1.1 and provide the essential components necessary for the reproducibility each project's results, as described in Section 4.5. All benchmark runs can be conveniently accessed at https://github.com/kchousos/OverHAuL/actions/workflows/benchmarks.yml.

[1] https://kchousos.github.io/OverHAuL/

8. Discussion

As discussed in Section Chapter 6, 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 [117], [118].

8.1. 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 that LLMs could hallucinate 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.

9. 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.

9.1. 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 [93], [94], [119], [120]—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++ [30] could diversify the fuzzing strategies available to OverHAuL, while exploring more experimental tools such as GraphFuzz [68] 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 [121]. 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.

9.2. 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 [54]–[56], 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.

9.3. 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 [95], 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], [66].

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

9.4. 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.

10. Conclusion

Recap

Performed a literature review of similar projects.

Presented the algorithm *and* the implementation.

generative AI disclaimer à la ACM?

Disclaimer on GenAI

During the preparation of this work, the authors used OpenAI GPT-4o for *grammar and spelling check* and simple rephrasing. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

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. (Dec. 18, 2023), [Online]. Available: http://arxiv.org/abs/2211.03622, pre-published.

[7] N. Kosmyna, 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 [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. [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. [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]. (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 Vulnerabilty 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. [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. 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/.

[30] M. Heuse, H. Eißfeldt, A. Fioraldi, and D. Maier, *AFL++*, version 4.00c, Jan. 2022. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus.

[31] LLVM Project. "The LLVM Compiler Infrastructure Project." (2025), [Online]. Available: https://llvm.org/.

[32] F. Bellard, P. Maydell, and QEMU Team, *QEMU*, version 10.0.2, May 29, 2025. [Online]. Available: https://www.qemu.org/.

[33] Unicorn Engine, *Unicorn-engine/unicorn*, Unicorn Engine, Jul. 15, 2025. [Online]. Available: https://github.com/unicorn-engine/unicorn.

[34] 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. [Online]. Available: https://dl.acm.org/doi/10.1145/3490443.

[35] 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. [Online]. Available: https://doi.org/10.1007/s43681-024-00583-7.

[36] D. Bahdanau, K. Cho, and Y. Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate." arXiv: 1409.0473 [cs, stat]. (May 19, 2016), [Online]. Available: http://arxiv.org/abs/1409.0473, pre-published.

[37] A. Vaswani, N. Shazeer, N. Parmar, *et al.* "Attention Is All You Need." arXiv: 1706.03762 [cs]. (Aug. 1, 2023), [Online]. Available: http://arxiv.org/abs/1706.03762, pre-published.

[38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." arXiv: 1810.04805 [cs]. (May 24, 2019), [Online]. Available: http://arxiv.org/abs/1810.04805, pre-published.

[39] 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.

[40] 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.

[41] T. B. Brown, B. Mann, N. Ryder, *et al.* "Language Models are Few-Shot Learners." arXiv: 2005.14165 [cs]. (Jul. 22, 2020), [Online]. Available: http://arxiv.org/abs/2005.14165, pre-published.

[42] OpenAI, J. Achiam, S. Adler, *et al.* "GPT-4 Technical Report." arXiv: 2303.08774 [cs]. (Mar. 4, 2024), [Online]. Available: http://arxiv.org/abs/2303.08774, pre-published.

[43] Anthropic. "Claude." (2025), [Online]. Available: https://claude.ai/new.

[44] DeepSeek-AI, D. Guo, D. Yang, *et al.* "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." arXiv: 2501.12948 [cs]. (Jan. 22, 2025), [Online]. Available: http://arxiv.org/abs/2501.12948, pre-published.

[45] A. Grattafiori, A. Dubey, A. Jauhri, *et al.* "The Llama 3 Herd of Models." arXiv: 2407.21783 [cs]. (Nov. 23, 2024), [Online]. Available: http://arxiv.org/abs/2407.21783, pre-published.

[46] OpenAI. "ChatGPT." (2025), [Online]. Available: https://chatgpt.com.

[47] Google. "Google Gemini," Gemini. (2025), [Online]. Available: https://gemini.google.com.

[48] J. Wei, X. Wang, D. Schuurmans, *et al.* "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." arXiv: 2201.11903 [cs]. (Jan. 10, 2023), [Online]. Available: http://arxiv.org/abs/2201.11903, pre-published.

[49] S. Yao, D. Yu, J. Zhao, *et al.* "Tree of Thoughts: Deliberate Problem Solving with Large Language Models." arXiv: 2305.10601 [cs]. (Dec. 3, 2023), [Online]. Available: http://arxiv.org/abs/2305.10601, pre-published.

[50] P. Lewis, E. Perez, A. Piktus, *et al.* "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." arXiv: 2005.11401 [cs]. (Apr. 12, 2021), [Online]. Available: http://arxiv.org/abs/2005.11401, pre-published.

[51] S. Yao, J. Zhao, D. Yu, *et al.* "ReAct: Synergizing Reasoning and Acting in Language Models." arXiv: 2210.03629. (Mar. 10, 2023), [Online]. Available: http://arxiv.org/abs/2210.03629, pre-published.

[52] Anysphere. "Cursor - The AI Code Editor." (2025), [Online]. Available: https://cursor.com/.

[53] Microsoft. "GitHub Copilot · Your AI pair programmer," GitHub. (2025), [Online]. Available: https://github.com/features/copilot.

[54] E. Nijkamp, B. Pang, H. Hayashi, *et al.*, "CodeGen: An open large language model for code with multi-turn program synthesis," *ICLR*, 2023.

[55] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "CodeGen2: Lessons for training llms on programming and natural languages," *ICLR*, 2023.

[56] OpenAI. "Introducing GPT-4.1 in the API." (Apr. 14, 2025), [Online]. Available: https://openai.com/index/gpt-4-1/.

[57] A. Sarkar and I. Drosos. "Vibe coding: Programming through conversation with artificial intelligence." arXiv: 2506.23253 [cs]. (Jun. 29, 2025), [Online]. Available: http://arxiv.org/abs/2506.23253, pre-published.

[58] A. Sheth, K. Roy, and M. Gaur. "Neurosymbolic AI – Why, What, and How." arXiv: 2305.00813 [cs]. (May 1, 2023), [Online]. Available: http://arxiv.org/abs/2305.00813, pre-published.

[59] A. d'Avila Garcez and L. C. Lamb. "Neurosymbolic AI: The 3rd Wave." arXiv: 2012.05876. (Dec. 16, 2020), [Online]. Available: http://arxiv.org/abs/2012.05876, pre-published.

[60] D. Ganguly, S. Iyengar, V. Chaudhary, and S. Kalyanaraman. "Proof of Thought : Neurosymbolic Program Synthesis allows Robust and Interpretable Reasoning." arXiv: 2409.17270. (Sep. 25, 2024), [Online]. Available: http://arxiv.org/abs/2409.17270, pre-published.

[61] M. Gaur and A. Sheth. "Building Trustworthy NeuroSymbolic AI Systems: Consistency, Reliability, Explainability, and Safety." arXiv: 2312.06798. (Dec. 5, 2023), [Online]. Available: http://arxiv.org/abs/2312.06798, pre-published.

[62] D. Tilwani, R. Venkataramanan, and A. P. Sheth. "Neurosymbolic AI approach to Attribution in Large Language Models." arXiv: 2410.03726. (Sep. 30, 2024), [Online]. Available: http://arxiv.org/abs/2410.03726, pre-published.

[63] 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. [Online]. Available: https://journals.sagepub.com/doi/full/10.3233/AIC-210084.

[64] 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.

[65] Z. Li, S. Dutta, and M. Naik. "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities." arXiv: 2405.17238 [cs]. (Apr. 6, 2025), [Online]. Available: http://arxiv.org/abs/2405.17238, pre-published.

[66] 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. [Online]. Available: https://doi.org/10.1145/3689236.3689272.

[67] 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 [cs]. (Sep. 1, 2024), [Online]. Available: http://arxiv.org/abs/2409.00922, pre-published.

[68] 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. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510228.

[69] 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 [cs]. (Apr. 4, 2023), [Online]. Available: http://arxiv.org/abs/2304.02014, pre-published.

[70] 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. [Online]. Available: https://dl.acm.org/doi/10.1145/3597926.3598067.

[71] 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.

[72] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu, *OSS-Fuzz*, Apr. 8, 2025. [Online]. Available: https://github.com/google/oss-fuzz.

[73] 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. [Online]. Available: http://www.airccse.org/journal/ijsea/papers/0711ijsea04.pdf.

[74] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang. "IntelliGen: Automatic Driver Synthesis for FuzzTesting." arXiv: 2103.00862 [cs]. (Mar. 1, 2021), [Online]. Available: http://arxiv.org/abs/2103.00862, pre-published.

[75] H. Xu, W. Ma, T. Zhou, *et al.* "CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph." arXiv: 2411.11532 [cs]. (Dec. 20, 2024), [Online]. Available: http://arxiv.org/abs/2411.11532, pre-published.

[76] Y. Lyu, Y. Xie, P. Chen, and H. Chen. "Prompt Fuzzing for Fuzz Driver Generation." arXiv: 2312.17677 [cs]. (May 29, 2024), [Online]. Available: http://arxiv.org/abs/2312.17677, pre-published.

[77] OSS-Fuzz. "OSS-Fuzz Documentation," OSS-Fuzz. (2025), [Online]. Available: https://google.github.io/oss-fuzz/.

[78] Google, *Google/clusterfuzz*, Google, Apr. 9, 2025. [Online]. Available: https://github.com/google/clusterfuzz.

[79] Google, *Google/fuzztest*, Google, Jul. 10, 2025. [Online]. Available: https://github.com/google/fuzztest.

[80] Google, *Google/honggfuzz*, Google, Jul. 10, 2025. [Online]. Available: https://github.com/google/honggfuzz.

[81] 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.

[82] Open Source Security Foundation (OpenSSF), *Ossf/fuzz-introspector*, Open Source Security Foundation (OpenSSF), Jun. 30, 2025. [Online]. Available: https://github.com/ossf/fuzz-introspector.

[83] L. Thomason, *Leethomason/tinyxml2*, Jul. 10, 2025. [Online]. Available: https://github.com/leethomason/tinyxml2.

[84] 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/.

[85] L. Torvalds, *Git*, Apr. 7, 2005. [Online]. Available: https://git-scm.com/.

[86] T. Mikolov, K. Chen, G. Corrado, and J. Dean. "Efficient Estimation of Word Representations in Vector Space." arXiv: 1301.3781 [cs]. (Sep. 6, 2013), [Online]. Available: http://arxiv.org/abs/1301.3781, pre-published.

[87] D. A. Wheeler. "Flawfinder Home Page," Flawfinder. (), [Online]. Available: https://dwheeler.com/flawfinder/.

[88] 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 [cs]. (Sep. 23, 2024), [Online]. Available: http://arxiv.org/abs/2409.14924, pre-published.

[89] M. Chen, J. Tworek, H. Jun, *et al.* "Evaluating Large Language Models Trained on Code." arXiv: 2107.03374 [cs]. (Jul. 14, 2021), [Online]. Available: http://arxiv.org/abs/2107.03374, pre-published.

[90] O. Khattab, A. Singhvi, P. Maheshwari, *et al.* "DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines." arXiv: 2310.03714 [cs]. (Oct. 5, 2023), [Online]. Available: http://arxiv.org/abs/2310.03714, pre-published.

[91] Python Software Foundation. "Venv — Creation of virtual environments," Python documentation. (Jul. 17, 2025), [Online]. Available: https://docs.python.org/3/library/venv.html.

[92] pip developers. "Pip documentation v25.1.1." (2025), [Online]. Available: https://pip.pypa.io/en/stable/.

[93] A. Cedilnik, B. Hoffman, B. King, K. Martin, and A. Neundorf, *CMake - Upgrade Your Software Build System*, 2000. [Online]. Available: https://cmake.org/.

[94] 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. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402.

[95] Clibs Project. "Clib Packages," GitHub. (2025), [Online]. Available: https://github.com/clibs/clib/wiki/Packages.

[96] Clibs Project, *Clibs/clib*, clibs, Jul. 1, 2025. [Online]. Available: https://github.com/clibs/clib.

[97] 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.

[98] F. Bacon, *Of the Proficience and Advancement of Learning... Edited by the Rev. GW Kitchin.* Bell & Daldy, 1861.

[99] T. Preston-Werner. "Semantic Versioning 2.0.0," Semantic Versioning. (), [Online]. Available: https://semver.org/.

[100] OpenAI Docs. "GPT-4.1 mini - Open AI API." (2025), [Online]. Available: https://platform.openai.com.

[101] 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.

[102] T. He, *Sighingnow/libclang*, Jul. 3, 2025. [Online]. Available: https://github.com/sighingnow/libclang.

[103] OpenAI Docs. "Text-embedding-3-small - OpenAI API." (2025), [Online]. Available: https://platform.openai.com.

[104] M. Douze, A. Guzhva, C. Deng, *et al.* "The Faiss library." arXiv: 2401.08281 [cs]. (Feb. 11, 2025), [Online]. Available: http://arxiv.org/abs/2401.08281, pre-published.

[105] Stanford NLP Team. "Signatures - DSPy Documentation." (2025), [Online]. Available: https://dspy.ai/learn/programming/signatures/.

[106] Stanford NLP Team. "ReAct - DSPy Documentation." (2025), [Online]. Available: https://dspy.ai/api/modules/ReAct/.

[107] H. Chase, *LangChain*, Oct. 2022. [Online]. Available: https://github.com/langchain-ai/langchain.

[108] J. Liu, *LlamaIndex*, Nov. 2022. DOI: 10.5281/zenodo.1234. [Online]. Available: https://github.com/jerryjliu/llama_index.

[109] 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.

[110] M. Woolf. "The Problem With LangChain." (Jul. 14, 2023), [Online]. Available: https://minimaxir.com/2023/07/langchain-problem/.

[111] Woyera. "6 Reasons why Langchain Sucks," Medium. (Sep. 8, 2023), [Online]. Available: https://medium.com/@woyera/6-reasons-why-langchain-sucks-b6c99c98efbe.

[112] Astral, *Astral-sh/uv*, Astral, Jul. 18, 2025. [Online]. Available: https://github.com/astral-sh/uv.

[113] Astral, *Astral-sh/ruff*, Astral, Jul. 18, 2025. [Online]. Available: https://github.com/astral-sh/ruff.

[114] A. Cortesi, M. Hils, and T. Kriechbaumer, *Mitmproxy/pdoc*, mitmproxy, Jul. 18, 2025. [Online]. Available: https://github.com/mitmproxy/pdoc.

[115] PyTest Dev Team, *Pytest-dev/pytest*, pytest-dev, Jul. 18, 2025. [Online]. Available: https://github.com/pytest-dev/pytest.

[116] Python Software Foundation, *Python/mypy*, Python, Jul. 18, 2025. [Online]. Available: https://github.com/python/mypy.

[117] OpenAI Docs. "Model optimization - OpenAI API." (2025), [Online]. Available: https://platform.openai.com.

[118] 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.

[119] E. Martin, *Ninja-build/ninja*, ninja-build, Jul. 14, 2025. [Online]. Available: https://github.com/ninja-build/ninja.

[120] J. Pakkanen, *Mesonbuild/meson*, The Meson Build System, Jul. 14, 2025. [Online]. Available: https://github.com/mesonbuild/meson.

[121] 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 4.2.1) 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 [48]. 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. DSPy Custom Signatures

```python
class GenerateHarness(dspy.Signature):
    """
    You are an experienced C/C++ security testing engineer. You must write a
    libFuzzer-compatible `int LLVMFuzzerTestOneInput(const uint8_t *data, size_t
    size)` harness for a function of the given C project. Your goal is for the
    harness to be ready for compilation and for it to find successfully a bug in
    the function-under-test. Write verbose (within reason) and helpful comments
    on each step/decision you take/make, especially if you use "weird" constants
    or values that have something to do with the project.

    You have access to a rag_tool, which contains a vector store of
    function-level chunks of the project. Use it to write better harnesses. Keep
    in mind that it can only reply with function chunks, do not ask it to
    combine stuff.

    The rag_tool does not store any information on which lines the functions
    are. So do not ask questions based on lines.

    Make sure that you only fuzz an existing function. You will know that a
    functions exists when the rag_tool returns to you its signature and body.
    """

    static: str = dspy.InputField(
        desc=""" Output of static analysis tools for the project. If you find it
        helpful, write your harness so that it leverages some of the potential
        vulnerabilities described below.  """
    )
    new_harness: str = dspy.OutputField(
        desc=""" C code for a libFuzzer-compatible harness. Output only the C
        code, **DO NOT format it in a markdown code cell with backticks**, so
        that it will be ready for compilation.

        <important>

        Add **all** the necessary includes, either project-specific or standard
```

```
        libraries like <string.h>, <stdint.h> and <stdlib.h>. Also include any
        header files that are part of the project and are probably useful. Most
        projects have a header file with the same name as the project at the
        root.

        **The function to be fuzzed absolutely must be part of the source
        code**, do not write a harness for your own functions or speculate about
        existing ones. You must be sure that the function that is fuzzed exists
        in the source code. Use your rag tool to query the source code.

        Do not try to fuzz functions of the project that are static, since they
        are only visible in the file that they were declared. Choose other
        user-facing functions instead.

        </important>

        **Do not truncate the input to a smaller size that the original**,
        e.g. for avoiding large stack usage or to avoid excessive buffers. Opt
        to using the heap when possible to increase the chance of exposing
        memory errors of the library, e.g. mmap instead of declaring
        buf[1024]. Any edge cases should be handled by the library itself, not
        the harness. On the other hand, do not write code that will most
        probably crash irregardless of the library under test. The point is for
        a function of the library under test to crash, not the harness
        itself. Use and take advantage of any custom structs that the library
        declares.

        Do not copy function declarations inside the harness. The harness will
        be compiled in the root directory of the project.  """
    )


class FixHarness(dspy.Signature):
    """
    You are an experienced C/C++ security testing engineer. Given a
    libFuzzer-compatible harness that fails to compile and its compilation
    errors, rewrite it so that it compiles successfully. Analyze the compilation
    errors carefully and find the root causes. Add any missing #includes like
    <string.h>, <stdint.h> and <stdlib.h> and #define required macros or
    constants in the fuzz target. If needed, re-declare functions or struct
    types. Add verbose comments to explain what you changed and why.
    """

```

```python
    old_harness: str = dspy.InputField(desc="The harness to be fixed.")
    error: str = dspy.InputField(desc="The compilaton error of the harness.")
    new_harness: str = dspy.OutputField(
        desc="""The newly created harness with the necessary modifications for
        correct compilation."""
    )


class ImproveHarness(dspy.Signature):
    f"""
    You are an experienced C/C++ security testing engineer. Given a
    libFuzzer-compatible harness that does not find any bug/does not crash (even
    after running for {Config.EXECUTION_TIMEOUT} seconds) or has memory leaks
    (generates leak files), you are called to rewrite it and improve it so that
    a bug can be found more easily and/or memory is managed correctly. Determine
    the information you need to write an effective fuzz target and understand
    constraints and edge cases in the source code to do it more
    effectively. Reply only with the source code --- without backticks.  Add
    verbose comments to explain what you changed and why.
    """

    old_harness: str = dspy.InputField(
        desc="The harness to be improved so it can find a bug more quickly."
    )
    output: str = dspy.InputField(desc="The output of the harness' execution.")
    new_harness: str = dspy.OutputField(
        desc="""The newly created harness with the necessary modifications for
        quicker bug-finding. If the provided harness has unnecessary input
        limitations regarding size or format etc., remove them."""
    )
```