

1

OverHAuL

2

Harnessing Automation for C Libraries via LLMs

3

Konstantinos Chousos

4

July, 2025

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

Preface

12 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sagittis posuere ligula sit amet lacinia.
13 Duis dignissim pellentesque magna, rhoncus congue sapien finibus mollis. Ut eu sem laoreet,
14 vehicula ipsum in, convallis erat. Vestibulum magna sem, blandit pulvinar augue sit amet, auctor
15 malesuada sapien. Nullam faucibus leo eget eros hendrerit, non laoreet ipsum lacinia. Curabitur
16 cursus diam elit, non tempus ante volutpat a. Quisque hendrerit blandit purus non fringilla. Integer
17 sit amet elit viverra ante dapibus semper. Vestibulum viverra rutrum enim, at luctus enim posuere
18 eu. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

19 Nunc ac dignissim magna. Vestibulum vitae egestas elit. Proin feugiat leo quis ante condimentum,
20 eu ornare mauris feugiat. Pellentesque habitant morbi tristique senectus et netus et malesuada fames
21 ac turpis egestas. Mauris cursus laoreet ex, dignissim bibendum est posuere iaculis. Suspendisse
22 et maximus elit. In fringilla gravida ornare. Aenean id lectus pulvinar, sagittis felis nec, rutrum
23 risus. Nam vel neque eu arcu blandit fringilla et in quam. Aliquam luctus est sit amet vestibulum
24 eleifend. Phasellus elementum sagittis molestie. Proin tempor lorem arcu, at condimentum purus
25 volutpat eu. Fusce et pellentesque ligula. Pellentesque id tellus at erat luctus fringilla. Suspendisse
26 potenti.

27 Etiam maximus accumsan gravida. Maecenas at nunc dignissim, euismod enim ac, bibendum ipsum.
28 Maecenas vehicula velit in nisl aliquet ultricies. Nam eget massa interdum, maximus arcu vel,
29 pretium erat. Maecenas sit amet tempor purus, vitae aliquet nunc. Vivamus cursus urna velit,
30 eleifend dictum magna laoreet ut. Duis eu erat mollis, blandit magna id, tincidunt ipsum. Integer
31 massa nibh, commodo eu ex vel, venenatis efficitur ligula. Integer convallis lacus elit, maximus
32 eleifend lacus ornare ac. Vestibulum scelerisque viverra urna id lacinia. Vestibulum ante ipsum
33 primis in faucibus orci luctus et ultrices posuere cubilia curae; Aenean eget enim at diam bibendum
34 tincidunt eu non purus. Nullam id magna ultrices, sodales metus viverra, tempus turpis.

35 Acknowledgments

36 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sagittis posuere ligula sit amet lacinia.
37 Duis dignissim pellentesque magna, rhoncus congue sapien finibus mollis. Ut eu sem laoreet,
38 vehicula ipsum in, convallis erat. Vestibulum magna sem, blandit pulvinar augue sit amet, auctor
39 malesuada sapien. Nullam faucibus leo eget eros hendrerit, non laoreet ipsum lacinia. Curabitur
40 cursus diam elit, non tempus ante volutpat a. Quisque hendrerit blandit purus non fringilla. Integer
41 sit amet elit viverra ante dapibus semper. Vestibulum viverra rutrum enim, at luctus enim posuere
42 eu. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Table of contents

1	Introduction	1
1.1	Motivation	1
1.2	Preview of following sections (rename)	2
2	Background	3
2.1	Fuzz Testing	3
2.1.1	Motivation	4
2.1.2	Methodology	5
2.1.3	Challenges in Adoption	7
2.2	Large Language Models	7
2.2.1	Biggest GPTs	8
2.2.2	Prompting	8
2.2.3	LLMs for Coding	9
2.2.4	LLMs for Fuzzing	10
2.3	Neurosymbolic AI	10
3	Related work	12
3.1	Previous Projects	12
3.1.1	KLEE	12
3.1.2	IRIS	12
3.1.3	FUDGE	13
3.1.4	UTopia	13
3.1.5	FuzzGen	13
3.1.6	IntelliGen	14
3.1.7	CKGFuzzer	14
3.1.8	PromptFuzz	15
3.1.9	OSS-Fuzz	16
3.1.10	OSS-Fuzz-Gen	16
3.1.11	AutoGen	17
3.2	Differences	17
4	OverHAuL	19
4.1	Architecture	20
4.1.1	Project Analysis	20
4.1.2	Harness Creation	21
4.1.3	Harness Evaluation	21
4.2	Main techniques	21
4.3	Examples	23

79	4.4	Scope	23
80	4.4.1	Assumptions/Prerequisites	24
81	4.5	Abandoned techniques	24
82	5	Evaluation	25
83	5.1	Research questions	25
84	5.2	Benchmarks	25
85	5.3	Performance	25
86	5.4	Issues	25
87	5.5	Future Work	25
88	5.5.1	Technical Future Work	25
89	5.5.2	Architectural Future Work/Extensions	25
90	6	Results	26
91	7	Implementation	27
92	7.1	Development environment	27
93	7.2	Reproducibility	27
94	8	Future Work	28
95	8.1	Enhancements to Core Features	28
96	8.2	Experimentation with Large Language Models and Data Representation	29
97	8.3	Comprehensive Evaluation and Benchmarking	29
98	8.4	Practical Deployment and Community Engagement	29
99	9	Discussion	31
100	10	Conclusion	32
101		Bibliography	33

1 Introduction

103 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sagittis posuere ligula sit amet lacinia.
104 Duis dignissim pellentesque magna, rhoncus congue sapien finibus mollis. Ut eu sem laoreet,
105 vehicula ipsum in, convallis erat. Vestibulum magna sem, blandit pulvinar augue sit amet, auctor
106 malesuada sapien. Nullam faucibus leo eget eros hendrerit, non laoreet ipsum lacinia. Curabitur
107 cursus diam elit, non tempus ante volutpat a. Quisque hendrerit blandit purus non fringilla. Integer
108 sit amet elit viverra ante dapibus semper. Vestibulum viverra rutrum enim, at luctus enim posuere
109 eu. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

110 Nunc ac dignissim magna. Vestibulum vitae egestas elit. Proin feugiat leo quis ante condimentum,
111 eu ornare mauris feugiat. Pellentesque habitant morbi tristique senectus et netus et malesuada fames
112 ac turpis egestas. Mauris cursus laoreet ex, dignissim bibendum est posuere iaculis. Suspendisse
113 et maximus elit. In fringilla gravida ornare. Aenean id lectus pulvinar, sagittis felis nec, rutrum
114 risus. Nam vel neque eu arcu blandit fringilla et in quam. Aliquam luctus est sit amet vestibulum
115 eleifend. Phasellus elementum sagittis molestie. Proin tempor lorem arcu, at condimentum purus
116 volutpat eu. Fusce et pellentesque ligula. Pellentesque id tellus at erat luctus fringilla. Suspendisse
117 potenti.

1.1 Motivation

119 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sagittis posuere ligula sit amet lacinia.
120 Duis dignissim pellentesque magna, rhoncus congue sapien finibus mollis. Ut eu sem laoreet,
121 vehicula ipsum in, convallis erat. Vestibulum magna sem, blandit pulvinar augue sit amet, auctor
122 malesuada sapien. Nullam faucibus leo eget eros hendrerit, non laoreet ipsum lacinia. Curabitur
123 cursus diam elit, non tempus ante volutpat a. Quisque hendrerit blandit purus non fringilla. Integer
124 sit amet elit viverra ante dapibus semper. Vestibulum viverra rutrum enim, at luctus enim posuere
125 eu. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

126 Nunc ac dignissim magna. Vestibulum vitae egestas elit. Proin feugiat leo quis ante condimentum,
127 eu ornare mauris feugiat. Pellentesque habitant morbi tristique senectus et netus et malesuada fames
128 ac turpis egestas. Mauris cursus laoreet ex, dignissim bibendum est posuere iaculis. Suspendisse
129 et maximus elit. In fringilla gravida ornare. Aenean id lectus pulvinar, sagittis felis nec, rutrum
130 risus. Nam vel neque eu arcu blandit fringilla et in quam. Aliquam luctus est sit amet vestibulum
131 eleifend. Phasellus elementum sagittis molestie. Proin tempor lorem arcu, at condimentum purus
132 volutpat eu. Fusce et pellentesque ligula. Pellentesque id tellus at erat luctus fringilla. Suspendisse
133 potenti.

134 [1]

135 1.2 Preview of following sections (rename)

136 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis sagittis posuere ligula sit amet lacinia.
137 Duis dignissim pellentesque magna, rhoncus congue sapien finibus mollis. Ut eu sem laoreet,
138 vehicula ipsum in, convallis erat. Vestibulum magna sem, blandit pulvinar augue sit amet, auctor
139 malesuada sapien. Nullam faucibus leo eget eros hendrerit, non laoreet ipsum lacinia. Curabitur
140 cursus diam elit, non tempus ante volutpat a. Quisque hendrerit blandit purus non fringilla. Integer
141 sit amet elit viverra ante dapibus semper. Vestibulum viverra rutrum enim, at luctus enim posuere
142 eu. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

2 Background

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 [2]. In essence, 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 [3]–[5].

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” [6]. 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 [2]:

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.

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

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

178 In practice, bug oracles often rely on runtime instrumentation techniques, such as monitoring for
179 fatal POSIX signals (e.g., SIGSEGV) or using sanitizers like AddressSanitizer (ASan) [7]. Tools like
180 LibFuzzer [8] commonly incorporate such instrumentation to reliably identify crashes or memory
181 errors during fuzzing.

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

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

188 The process of selecting an effective initial corpus is crucial because it directly impacts how quickly
189 and thoroughly the fuzzer can cover the target program’s code. This challenge—studied as the
190 *seed-selection problem*—involves identifying seeds that enable rapid discovery of diverse execution
191 paths and is non-trivial [9]. A well-chosen seed set often accelerates bug discovery and improves
192 overall fuzzing efficiency.

193 2.1.1 Motivation

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

197 — OWASP Foundation [10]

198 Fuzz testing offers several tangible benefits:

- 199 1. **Early vulnerability discovery:** Detecting defects during development is cheaper and safer
200 than addressing exploits in production.
- 201 2. **Adversary-parity:** Performing the same randomised stress that attackers employ allows
202 defenders to pre-empt zero-days.
- 203 3. **Robustness and correctness:** Beyond security, fuzzing exposes logic errors and stability
204 issues in complex, high-throughput APIs (e.g., decompressors) even when inputs are *expected*
205 to be well-formed.
- 206 4. **Regression prevention:** Re-running a corpus of crashing inputs as part of continuous
207 integration ensures that fixed bugs remain fixed.

2.1.1.1 Success Stories

Heartbleed (CVE-2014-0160) [11], [12] arose from a buffer over-read¹ in OpenSSL [13] introduced on 1 February 2012 and unnoticed until 1 April 2014. Post-mortem analyses showed that a simple fuzz campaign exercising the TLS heartbeat extension would have revealed the defect almost immediately [14].

Likewise, the *Shellshock* (or *Bashdoor*) family of bugs in GNU Bash [15] 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 [16] in late 2014 [17].

On the defensive tooling side, the security tool named *Mayhem*—developed by the company of the same name—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 [18].

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 program under test (PUT) is typically conducted using a dedicated fuzzing engine (see Definition 2.3). Among the most widely adopted fuzzers for C and C++ projects and libraries are AFL [16]—which has since evolved into AFL++ [19]—and LibFuzzer [8]. Within the OverHAuL framework, LibFuzzer is preferred owing to its superior suitability for library fuzzing, whereas AFL++ predominantly targets executables and binary fuzzing.

2.1.2.1 LibFuzzer

LibFuzzer [8] is an in-process, coverage-guided evolutionary fuzzing engine primarily designed for testing libraries. It forms part of the LLVM ecosystem [20] 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*.

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 [8]. 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 following function signature:

¹<https://xkcd.com/1354/>

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

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

242 A more illustrative example of such a harness is provided in Listing 2.2.

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

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

243 To compile and link such a harness with LibFuzzer, the Clang compiler—also part of the LLVM
244 project [20]—must be used alongside appropriate compiler flags. For instance, compiling the harness
245 in Listing 2.2 can be achieved as follows:

246 2.1.2.2 AFL and AFL++

247 *American Fuzzy Lop* (AFL) [16], developed by Michał Zalewski, is a seminal fuzzer targeting C and
248 C++ applications. Its core methodology relies on instrumented binaries to provide edge coverage
249 feedback, thereby guiding input mutation towards unexplored program paths. AFL supports several
250 emulation backends including QEMU [21]—an open-source CPU emulator facilitating fuzzing on
251 diverse architectures—and Unicorn [22], a lightweight multi-platform CPU emulator. While AFL
252 established itself as a foundational tool within the fuzzing community, its successor AFL++ [19]
253 incorporates numerous enhancements and additional features to improve fuzzing efficacy.

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

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

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

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

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

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

266 2.1.3 Challenges in Adoption

267 Despite its potential for uncovering software vulnerabilities, fuzzing remains a relatively underuti-
268 lized testing technique compared to more established methodologies such as Test-Driven Develop-
269 ment (TDD). This limited adoption can be attributed, in part, to the substantial initial investment
270 required to design and implement appropriate test harnesses that enable effective fuzzing processes.
271 Furthermore, the interpretation of fuzzing outcomes—particularly the identification, diagnostic
272 analysis, and prioritization of program crashes—demands considerable resources and specialized
273 expertise. These factors collectively pose significant barriers to the widespread integration of
274 fuzzing within standard software development and testing practices.

275 2.2 Large Language Models

276 Natural Language Processing (NLP), a subfield of Artificial Intelligence (AI), has a rich and ongoing
277 history that has evolved significantly since its beginning in the 1990s [23], [24]. Among the most
278 notable—and recent—advancements in this domain are Large Language Models (LLMs), which have
279 transformed the landscape of NLP and AI in general.

280 At the core of many LLMs is the attention mechanism, which was introduced by Bahdanau et
281 al. in 2014 [25]. This pivotal innovation enabled models to focus on relevant parts of the input
282 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 [26]. 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 [27]. 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 [28], further pushed the boundaries. Subsequent iterations, including GPT-2 [29], GPT-3 [30], and the most current GPT-4 [31], 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) [32]–[34]. 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 [32], [35], [36] represent commercially developed systems often optimized for specific tasks without public access to their underlying weights. In contrast, open-source models², including the Llama series [34] and Deepseek [33], 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 [30].

To enhance performance on more complex tasks, several advanced prompting techniques have emerged. One notable strategy is the *Chain of Thought* approach [37], 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 [38], which enables the LLM

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

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) [39] 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 [40], 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 [41], [42]. 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 [43]–[45].

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 [46]. 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 [47], [48]. 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.

394 At its core, NeSy AI facilitates the development of AI systems that are capable of understanding
395 and interpreting feedback in real-world scenarios [49]. This characteristic is particularly significant
396 in the current landscape of artificial intelligence, where LLMs are predominant. In this context,
397 NeSy AI is increasingly viewed as a critical solution to pressing issues related to explainability,
398 attribution, and reliability in AI systems [50], [51]. These challenges are essential for ensuring
399 that AI systems can be trusted and effectively utilized in various applications, from business to
400 healthcare.

401 The burgeoning field of neurosymbolic AI is still in its nascent stages, with ongoing research and
402 development actively exploring its potential to enhance attribution methodologies within large
403 language models. By addressing these critical challenges, NeSy AI can significantly contribute to
404 the broader landscape of trustworthy AI systems, allowing for more transparent and accountable
405 decision-making processes [47], [50], [51].

406 Moreover, the application of neurosymbolic AI within the domain of fuzzing is gaining traction,
407 paving the way for innovative explorations. This integration of LLMs with symbolic systems opens
408 up new avenues for research. Currently, there are only a limited number of tools that support such
409 hybrid approaches (Chapter 3). Among these, OverHAuL constitutes a Neuro[Symbolic] tool, as
410 classified by Henry Kautz’s taxonomy [52], [53]. This means that the neural model—specifically the
411 LLM—can leverage symbolic reasoning tools—in this case a source code explorer (Chapter 7)—to
412 augment its reasoning capabilities. This symbiotic relationship enhances the overall efficacy and
413 versatility of LLMs for fuzzing harnesses generation, demonstrating the profound potential held by
414 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 [26] and the 2020’s boom of commercial GPTs [35], 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 [54]–[63].

3.1 Previous Projects

3.1.1 KLEE

KLEE [64] 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 [20] 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 [54] 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 [63] 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 [65]. 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** [66], per FUDGE, based on its Abstract Syntax Tree (AST) using LLVM’s Clang tool [20]. 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 [59] (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.* [59], 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 [62], 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

483 builds an A²DG to record the API usages and infers its intended structure. After completion, this
484 directed graph contains all the valid API call sequences found in the client code corpus. It is built
485 in a two-step process: First, many smaller A²DGs are created, one for each root function per client
486 code snippet. Once such graphs have been created for all the available client code instances, they
487 are combined to formulate the master A²DG. This graph can be seen as a template for correct usage
488 of the library. **3. Fuzzer generator.** Through the A²DG, a fuzzing harness is created. Contrary to
489 FUDGE, FuzzGen does not create multiple “simple” harnesses but a single complex one with the
490 goal of covering the whole of the A²DG. In other words, while FUDGE fuzzes a single API call at a
491 time, FuzzGen’s result is a single harness that tries to fuzz the given library all at once through
492 complex API usage.

493 3.1.6 IntelliGen

494 SAMPLE

495 Zhang et al. present IntelliGen [67], a system for automatically synthesizing fuzz drivers by statically
496 identifying potentially vulnerable entry-point functions within C projects. Implemented using
497 LLVM, IntelliGen focuses on improving fuzzing efficiency by targeting code more likely to contain
498 memory safety issues, rather than exhaustively fuzzing all available functions.

499 The system comprises two main components: the **Entry Function Locator** and the **Fuzz Driver**
500 **Synthesizer**. The Entry Function Locator analyzes the project’s abstract syntax tree (AST) and clas-
501 sifies functions based on heuristics that indicate vulnerability. These include pointer dereferencing,
502 calls to memory-related functions (e.g., memcpy, memset), and invocation of other internal functions.
503 Functions that score highly on these metrics are prioritized for fuzz driver generation. The guiding
504 insight is that entry points with fewer argument checks and more direct memory operations expose
505 more useful program logic for fuzz testing.

506 The Fuzz Driver Synthesizer then generates harnesses for these entry points. For each target
507 function, it synthesizes a LLVMFuzzerTestOneInput function that invokes the target with arguments
508 derived from the fuzzer input. This process involves inferring argument types from the source code
509 and ensuring that runtime behavior does not violate memory safety—thus avoiding invalid inputs
510 that would cause crashes unrelated to genuine bugs.

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

515 3.1.7 CKGFuzzer

516 SAMPLE

517 CKGFuzzer [68] is a fuzzing framework designed to automate the generation of effective fuzz drivers
518 for C/C++ libraries by leveraging static analysis and large language models. Its workflow begins by
519 parsing the target project along with any associated library APIs to construct a code knowledge
520 graph. This involves two primary steps: first, parsing the abstract syntax tree (AST), and second,

521 performing interprocedural program analysis. Through this process, CKGFuzzer extracts essential
522 program elements such as data structures, function signatures, function implementations, and call
523 relationships.

524 Using the knowledge graph, CKGFuzzer then identifies and queries meaningful API combinations,
525 focusing on those that are either frequently invoked together or exhibit functional similarity.
526 It generates candidate fuzz drivers for these combinations and attempts to compile them. Any
527 compilation errors encountered during this phase are automatically repaired using heuristics and
528 domain knowledge. A dynamically updated knowledge base, constructed from prior library usage
529 patterns, guides both the generation and repair processes.

530 Once the drivers are successfully compiled, CKGFuzzer executes them while monitoring code
531 coverage at the file level. It uses coverage feedback to iteratively mutate underperforming API
532 combinations, refining them until new execution paths are discovered or a preset mutation budget
533 is exhausted.

534 Finally, any crashes triggered during fuzzing are subjected to a reasoning process based on chain-
535 of-thought prompting. To help determine their severity and root cause, CKGFuzzer consults an
536 LLM-generated knowledge base containing real-world examples of vulnerabilities mapped to known
537 Common Weakness Enumeration (CWE) entries.

538 3.1.8 PromptFuzz

539 SAMPLE

540 Lyu et al. (2024) introduce PromptFuzz [69], a system for automatically generating fuzz drivers using
541 LLMs, with a novel focus on **prompt mutation** to improve coverage. The system is implemented
542 in Rust and targets C libraries, aiming to explore more of the API surface with each iteration.

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

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

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

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

3.1.9 OSS-Fuzz

OSS-Fuzz [65], [70] 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 [11], [12], [14]. 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 [71] 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 [8], [19], [72], [73]. C, C++, Rust, Go, Python and Java/JVM projects are supported.

3.1.10 OSS-Fuzz-Gen

OSS-Fuzz-Gen (OFG) [57], [74] 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 [75]. 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 [76], line coverage went from 38% to 69% without any manual interventions [74].

In 2024, OSS-Fuzz-Gen introduced an experimental feature for generating harnesses in previously unfuzzed projects [77]. The code for this feature resides in the `experimental/from_scratch` directory of the project’s GitHub repository [57], with the latest known working commit being 171aac2 and the latest overall commit being four months ago.

3.1.11 AutoGen

AutoGen [55] 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.

3.2 Differences

OverHAuL differs, in some way, with each of the aforementioned works. Firstly, although KLEE and IRIS [54], [64] 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 [59], [62], [63], 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 [63]: “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 [57] 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 [65], 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 [71] 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 [70]: “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.

637 No feedback is utilized nor is there graceful error handling in the case of a step’s failure. Even
638 in the case of the experimental feature for bootstrapping unfuzzed projects, OFG’s performance
639 varies heavily. During experimentation, a lot of generated harnesses were still wrapped either in
640 Markdown backticks or `<code>` tags, or were accompanied with explanations inside the generated
641 .c source file. Even if code was formatted correctly, in many cases it missed necessary headers for
642 compilation or used undeclared functions.

643 Lastly, the closest counterpart to OverHAuL is AutoGen [55]. Their similarity stands in the
644 implementation of a feedback loop between LLM and generated harness. However, most other
645 implementation decisions remain distinct. One difference regards the fuzzed function. While
646 AutoGen requires a target function to be specified by the user in which it narrows during its
647 whole run, OverHAuL delegates this to the LLM, letting it explore the codebase and decide by
648 itself the best candidate. Another difference lies in the need—and the lack of—of documentation.
649 While AutoGen requires it to gather information for the given function, OverHAuL leans into the
650 role of a developer by reading the related code and comments and thus avoiding any mismatches
651 between documentation and code. Finally, the LLMs’ input is built based on predefined prompt
652 templates, a technique also present in OSS-Fuzz-Gen. OverHAuL operates one abstraction level
653 higher, leveraging DSPy [78] for programming instead of prompting the LLMs used.

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

- 657 1. Implementing a feedback mechanism between harness generation, compilation, and evalua-
658 tion phases,
- 659 2. Using autonomous ReAct agents capable of codebase exploration,
- 660 3. Leveraging a vector store for code consumption and retrieval.

4 OverHAuL

In this thesis we present *OverHAuL* (**H**arness **A**utomation with **L**LMs), a neurosymbolic AI tool that automatically generates fuzzing harnesses for C libraries through LLM agents. In its core, OverHAuL is comprised by three LLM ReAct agents [40]—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 [79] 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.

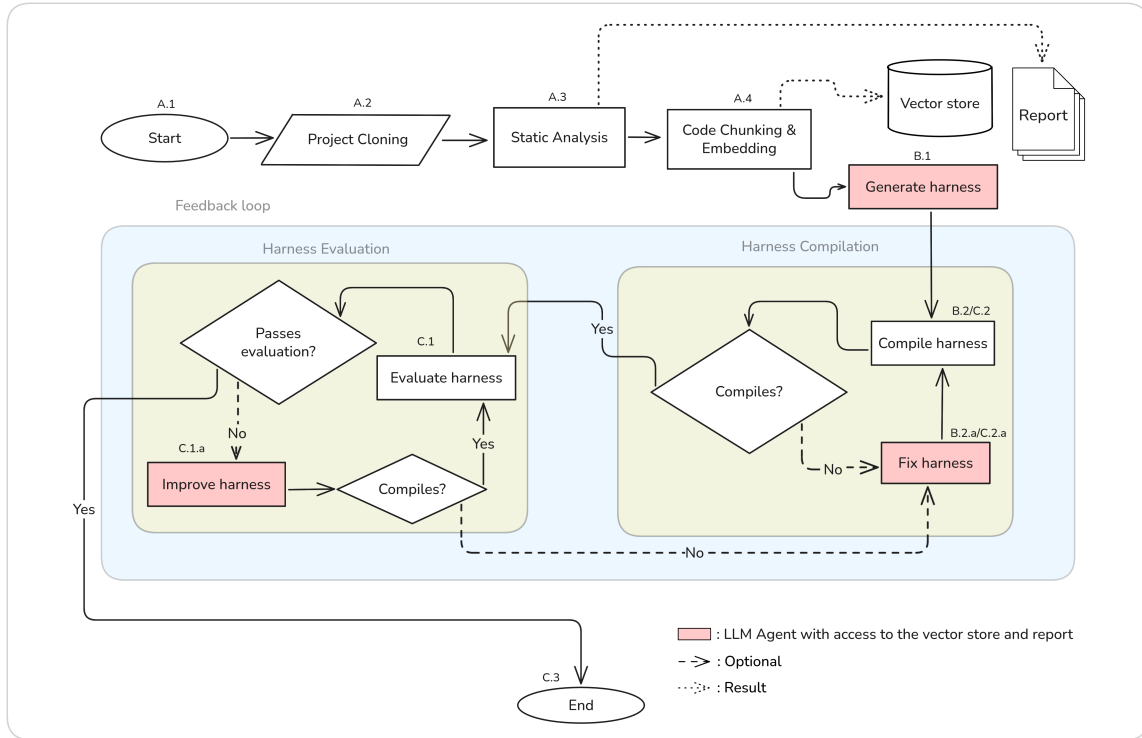


Figure 4.1: Overview of OverHAuL’s automatic harnessing process.

As commented in Section 3.2, OverHAuL does not expect and depend on the existence of client code or unit tests [59], [62], [63] nor does it require any preexisting fuzzing harnesses [57] or any documentation present [55]. Also importantly, OverHAuL is decoupled from other fuzzing projects, thus lowering the barrier to entry for new projects [57], [65]. 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 [40] which inspect and analyze the project’s source code. The latter is stored and interacted with as a set of text embeddings [80], 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 [81] 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 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.

¹I.e. “research code”.

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 $k = 3$ 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 4.4.1. 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

- Feedback loop (improvement + verification) This iterative structure creates a feedback loop which terminates if an acceptable working harness has been achieved or the *iteration budget* has been depleted. This iteration budget is shared among the compilation and evaluation loops, i.e. it is decreased any time a dashed arrow in the flowchart is followed.

- 745 • Triplet of ReAct agents An important point to stress is that each agent is a separate LLM
746 instance, albeit of the same model. This means that the agents have different and separate
747 contexts during each run. This technique allows more exploration per run, since for example
748 the “improver” agent can explore different search paths regarding the harness that the
749 “generator” agent may have internally rejected. It also keeps context window limitations
750 manageable, since the load is shared between the agents.
- 751 • Code exploration through vector store
 - 752 – Context window thoughts
 - 753 – Why RAG? Semantic chunking, most similar to paragraph-level chunking, most used
754 technique. Codex was fine-tuned with functions as input [82]. The bigger the chunk,
755 the bigger the noise [83].

756 The algorithm corresponding to the flowchart in Figure 4.1 can be seen in Algorithm 4.1.

Algorithm 4.1 Test Harness Generation with Iterative Refinement

```
1: procedure MAIN
2:   args  $\leftarrow$  PARSEARGUMENTS
3:   project_path, model  $\leftarrow$  args.project_path, args.model
4:   analyzer  $\leftarrow$  PROJECTANALYZER(project_path)
5:   project_info  $\leftarrow$  analyzer.COLLECTPROJECTINFO
6:   LOADENVIRONMENT
7:   embedder  $\leftarrow$  CREATEEMBEDDER(embedding_model)
8:   chunks  $\leftarrow$  EXTRACTCHUNKS(project_path)
9:   index, meta  $\leftarrow$  EMBEDCHUNKS(chunks, embedder)
10:  rag_tool  $\leftarrow$  MAKERAGTOOL(index, meta, embedder)
11:  harnesser  $\leftarrow$  HARNESSER(model, project_path, rag_tool)
12:  file_manager  $\leftarrow$  FILEMANAGER(project_path)
13:  builder  $\leftarrow$  HARNESSBUILDER(project_path)
14:  evaluator  $\leftarrow$  HARNESSEVALUATOR(project_path)
15:  acceptable  $\leftarrow$  false
16:  for i = 1 to MAX_ITERATIONS do
17:    harness  $\leftarrow$  harnesser.GENERATEHARNESS(project_info)
18:    file_manager.WRITEHARNESS(harness)
19:    error, compiled  $\leftarrow$  builder.BUILDHARNESS
20:    project_info.compiles  $\leftarrow$  compiled
21:    if  $\neg$ compiled then
22:      project_info.error  $\leftarrow$  error
23:      continue
24:    end if
25:    output, accepted  $\leftarrow$  evaluator.EVALUATEHARNESS
26:    if  $\neg$ accepted then
27:      project_info.output  $\leftarrow$  output
28:      continue
29:    else
30:      acceptable  $\leftarrow$  true
31:      break
32:    end if
33:  end for
34:  return acceptable  $\wedge$  project_info.compiles
35: end procedure
```

757 4.3 Examples

758 4.4 Scope

- 759 • Limited to C libraries
- 760 • Expects relatively simple project structure

```

} overhaul https://github.com/dvhar/dateparse -n gbt-4.1
2025-07-17 00:55:18.887 | INFO | overhaul.cliparse.arguments:184 - Cloning project's repo in the output/dateparse directory...
2025-07-17 00:55:18.976 | INFO | overhaul.core.harnesser._init_:167 - Initializing LLM...
2025-07-17 00:55:18.991 | INFO | overhaul.cli.main:243 - Iteration 1 of harnessing...
2025-07-17 00:55:18.991 | INFO | overhaul.core.harnesser.harness:207 - Calling LLM to generate a harness...
2025-07-17 00:55:22.837 | DEBUG | overhaul.rag.ragger:rag_tool:88 - Agent using RAG: "Show the functions in dateparse.c that contain a strcpy call, especially near lines 405, 800, and 1242."
2025-07-17 00:55:25.807 | DEBUG | overhaul.rag.ragger:rag_tool:88 - Agent using RAG: "Provide the full implementation (signature and body) of the dateparse function in dateparse.c."
2025-07-17 00:55:30.207 | DEBUG | overhaul.rag.ragger:rag_tool:88 - Agent using RAG: "Show the full signature and body of the function int dateparse(const char* datestr, date_t* t, int *offset, int stringlen) in dateparse.c."
2025-07-17 00:55:34.176 | DEBUG | overhaul.rag.ragger:rag_tool:88 - Agent using RAG: "Show the definition of struct parser and the typedef of date_t as used in dateparse.c and dateparse.h."
2025-07-17 00:55:38.581 | DEBUG | overhaul.rag.ragger:rag_tool:88 - Agent using RAG: "Show the definition of struct parser and the typedef of date_t from dateparse.h."
2025-07-17 00:56:01.897 | INFO | overhaul.io.file_manager.write_harness:60 - Writing harness to project...
2025-07-17 00:56:01.898 | INFO | overhaul.io.file_manager.write_harness:92 - Harness written to output/dateparse/harnesses/harness.c
2025-07-17 00:56:01.898 | INFO | overhaul.core.builder.build_harness:63 - Building harness...
2025-07-17 00:56:01.899 | INFO | overhaul.core.builder.build_harness:140 - Starting compilation of harness: harnesses/harness.c
2025-07-17 00:56:02.345 | INFO | overhaul.core.builder.build_harness:149 - Harness compiled successfully
2025-07-17 00:56:02.345 | INFO | overhaul.core.evaluator.evaluate_harness:81 - Evaluating harness...
2025-07-17 00:56:02.345 | INFO | overhaul.core.evaluator.evaluate_harness:90 - Starting execution of harness...
2025-07-17 00:56:02.417 | INFO | overhaul.core.evaluator.evaluate_harness:119 - Harness execution completed in 0.07 seconds.
2025-07-17 00:56:02.419 | INFO | overhaul.core.evaluator.evaluate_harness:181 - New testcases created (1): [{'crash-dfaa340e98889cd82d2cd680cf96fd04552a2b4', 1752702962.4113252}]
2025-07-17 00:56:02.419 | SUCCESS | overhaul.cli.main:282 - All done!

```

Figure 4.2: A successful execution of OverHAuL, harnessing the dateparse library.

- 761 – Code either in root or in common-named subdirs (e.g. src/)
- 762 – Any file or directory with main, test or example substring is ignored
- 763 – No main() function, or only exists in some file that is ignored by the above
- 764 • Build systems not supported
- 765 – Harness is compiled with a predefined command

766 4.4.1 Assumptions/Prerequisites

- 767 • Project structure
- 768 • file/folder naming
- 769 • building process

770 4.5 Abandoned techniques

- 771 1. Zero-shot harness generation
- 772 2. ChainOfThought modules for LLM instances [37]
- 773 3. Naive source code concatenation
- 774 4. manual {index, read}_tool usage for ReAct agents

775 5 Evaluation

776 5.1 Research questions

777 5.2 Benchmarks

778 10 open-source C/C++ projects.

779 5.3 Performance

780 5.4 Issues

781 5.5 Future Work

782 5.5.1 Technical Future Work

783 5.5.2 Architectural Future Work/Extensions

- 784 1. Build system
- 785 2. More (static) analysis tools integrations
- 786 3. General *localization* problem

787 6 Results

788 Results from integration with 10/100 open-source C/C++ projects.

789 7 Implementation

790 –depth 1 output/

791 embedder model openai Source code is processed and chunked using Clang [33]. The chunks are
792 function-level units, found to be a sweet-spot between semantic significance and size [82], [83].
793 This results in a list of Python dicts, each containing a function’s body, signature and filepath. Each
794 chunk’s function code is then turned into an embedding using OpenAI’s “text-embedding-3-small”
795 model. faiss store and index A FAISS [36] vector store is created. Each function embedding is stored
796 in it (with the same order, as to correspond with the previous list containing the metadata).

797 same order code chunks

798 Prompting techniques used (callback to Section 2.2.2). Sample prompt

799 [78]

800 1. Why instead of langchain or llamaindex? [84], [85]

801 libclang Python package

802 7.1 Development environment

803 uv, ruff, mypy, pytest, pdoc

804 7.2 Reproducibility

805 github workflow actions, artifacts, summary

8 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.

8.1 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 [86]–[89]—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++ [19] could diversify the fuzzing strategies available to OverHAuL, while exploring more experimental tools such as GraphFuzz [58] 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 [90]. 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 [62], [63], 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.

8.2 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 [43]–[45], 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.

8.3 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 [91], 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 [55], [57].

Complementing broad benchmarking, detailed ablation 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.

8.4 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.

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

883 9 Discussion

884 more powerful llms -> better results

885 open source libraries might have been in the training data results for closed source libraries could
886 be worse this could be mitigated with llm fine-tuning

887 **10 Conclusion**

888 Recap

889 generative AI disclaimer à la ACM?

Bibliography

- [1] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. “Do Users Write More Insecure Code with AI Assistants?” arXiv: [2211.03622](https://arxiv.org/abs/2211.03622). (Dec. 18, 2023), [Online]. Available: <http://arxiv.org/abs/2211.03622>, pre-published.
- [2] V. J. M. Manes, H. Han, C. Han, *et al.* “The Art, Science, and Engineering of Fuzzing: A Survey.” arXiv: [1812.00140](https://arxiv.org/abs/1812.00140) [cs]. (Apr. 7, 2019), [Online]. Available: <http://arxiv.org/abs/1812.00140>, pre-published.
- [3] 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.
- [4] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Upper Saddle River, NJ: Addison-Wesley, 2007, 543 pp., ISBN: 978-0-321-44611-4.
- [5] 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.
- [6] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1, 1990, ISSN: 0001-0782. DOI: [10.1145/96267.96279](https://doi.org/10.1145/96267.96279). [Online]. Available: <https://dl.acm.org/doi/10.1145/96267.96279>.
- [7] 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>.
- [8] LLVM Project. “libFuzzer – a library for coverage-guided fuzz testing. — LLVM 21.0.0git documentation.” (2025), [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>.
- [9] 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.
- [10] OWASP Foundation. “Fuzzing.” (), [Online]. Available: <https://owasp.org/www-community/Fuzzing>.
- [11] Blackduck, Inc. “Heartbleed Bug.” (Mar. 7, 2025), [Online]. Available: <https://heartbleed.com/>.
- [12] CVE Program. “CVE - CVE-2014-0160.” (2014), [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160>.
- [13] The OpenSSL Project, *Openssl/openssl*, OpenSSL, Jul. 15, 2025. [Online]. Available: <https://github.com/openssl/openssl>.
- [14] D. Wheeler. “How to Prevent the next Heartbleed.” (2014), [Online]. Available: <https://dwheeler.com/essays/heartbleed.html>.
- [15] GNU Project. “Bash - GNU Project - Free Software Foundation.” (), [Online]. Available: <https://www.gnu.org/software/bash/>.
- [16] M. Zalewski. “American fuzzy lop.” (), [Online]. Available: <https://lcamtuf.coredump.cx/afl/>.
- [17] 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>.

- [18] 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/>.
- [19] M. Heuse, H. Eißfeldt, A. Fioraldi, and D. Maier, *AFL++*, version 4.00c, Jan. 2022. [Online]. Available: <https://github.com/AFLplusplus/AFLplusplus>.
- [20] LLVM Project. “The LLVM Compiler Infrastructure Project.” (2025), [Online]. Available: <https://llvm.org/>.
- [21] F. Bellard, P. Maydell, and QEMU Team, *QEMU*, version 10.0.2, May 29, 2025. [Online]. Available: <https://www.qemu.org/>.
- [22] Unicorn Engine, *Unicorn-engine/unicorn*, Unicorn Engine, Jul. 15, 2025. [Online]. Available: <https://github.com/unicorn-engine/unicorn>.
- [23] H. Li, “Language models: Past, present, and future,” *Commun. ACM*, vol. 65, no. 7, pp. 56–63, Jun. 21, 2022, ISSN: 0001-0782. DOI: [10.1145/3490443](https://doi.org/10.1145/3490443). [Online]. Available: <https://dl.acm.org/doi/10.1145/3490443>.
- [24] Z. Wang, Z. Chu, T. V. Doan, S. Ni, M. Yang, and W. Zhang, “History, development, and principles of large language models: An introductory survey,” *AI and Ethics*, vol. 5, no. 3, pp. 1955–1971, Jun. 1, 2025, ISSN: 2730-5961. DOI: [10.1007/s43681-024-00583-7](https://doi.org/10.1007/s43681-024-00583-7). [Online]. Available: <https://doi.org/10.1007/s43681-024-00583-7>.
- [25] D. Bahdanau, K. Cho, and Y. Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate.” arXiv: [1409.0473](https://arxiv.org/abs/1409.0473) [cs, stat]. (May 19, 2016), [Online]. Available: <http://arxiv.org/abs/1409.0473>, pre-published.
- [26] A. Vaswani, N. Shazeer, N. Parmar, *et al.* “Attention Is All You Need.” arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs]. (Aug. 1, 2023), [Online]. Available: <http://arxiv.org/abs/1706.03762>, pre-published.
- [27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” arXiv: [1810.04805](https://arxiv.org/abs/1810.04805) [cs]. (May 24, 2019), [Online]. Available: <http://arxiv.org/abs/1810.04805>, pre-published.
- [28] 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>.
- [29] 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>.
- [30] T. B. Brown, B. Mann, N. Ryder, *et al.* “Language Models are Few-Shot Learners.” arXiv: [2005.14165](https://arxiv.org/abs/2005.14165) [cs]. (Jul. 22, 2020), [Online]. Available: <http://arxiv.org/abs/2005.14165>, pre-published.
- [31] OpenAI, J. Achiam, S. Adler, *et al.* “GPT-4 Technical Report.” arXiv: [2303.08774](https://arxiv.org/abs/2303.08774) [cs]. (Mar. 4, 2024), [Online]. Available: <http://arxiv.org/abs/2303.08774>, pre-published.
- [32] Anthropic. “Claude.” (2025), [Online]. Available: <https://claude.ai/new>.
- [33] DeepSeek-AI, D. Guo, D. Yang, *et al.* “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.” arXiv: [2501.12948](https://arxiv.org/abs/2501.12948) [cs]. (Jan. 22, 2025), [Online]. Available: [http://arxiv.org/abs/2501.12948](https://arxiv.org/abs/2501.12948), pre-published.
- [34] A. Grattafiori, A. Dubey, A. Jauhri, *et al.* “The Llama 3 Herd of Models.” arXiv: [2407.21783](https://arxiv.org/abs/2407.21783) [cs]. (Nov. 23, 2024), [Online]. Available: <http://arxiv.org/abs/2407.21783>, pre-published.
- [35] OpenAI. “ChatGPT.” (2025), [Online]. Available: <https://chatgpt.com>.
- [36] Google. “Google Gemini,” Gemini. (2025), [Online]. Available: <https://gemini.google.com>.
- [37] J. Wei, X. Wang, D. Schuurmans, *et al.* “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.” arXiv: [2201.11903](https://arxiv.org/abs/2201.11903) [cs]. (Jan. 10, 2023), [Online]. Available: <http://arxiv.org/abs/2201.11903>, pre-published.

- [38] 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.
- [39] 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.
- [40] 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.
- [41] Anysphere. “Cursor - The AI Code Editor.” (2025), [Online]. Available: <https://cursor.com/>.
- [42] Microsoft. “GitHub Copilot · Your AI pair programmer,” GitHub. (2025), [Online]. Available: <https://github.com/features/copilot>.
- [43] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, “CodeGen2: Lessons for training llms on programming and natural languages,” *ICLR*, 2023.
- [44] E. Nijkamp, B. Pang, H. Hayashi, *et al.*, “CodeGen: An open large language model for code with multi-turn program synthesis,” *ICLR*, 2023.
- [45] OpenAI. “Introducing Codex.” (May 16, 2025), [Online]. Available: <https://openai.com/index/introducing-codex/>.
- [46] 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.
- [47] 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.
- [48] 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.
- [49] 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.
- [50] 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.
- [51] 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.
- [52] 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>.
- [53] 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.
- [54] 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.
- [55] 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>.

- [56] 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.
- [57] 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>.
- [58] 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>.
- [59] 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>.
- [60] 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.
- [61] 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>.
- [62] 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>.
- [63] 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>.
- [64] 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>.
- [65] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu, *OSS-Fuzz*, Apr. 8, 2025. [Online]. Available: <https://github.com/google/oss-fuzz>.
- [66] 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>.
- [67] 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.
- [68] 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.
- [69] 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.

- [70] OSS-Fuzz. “OSS-Fuzz Documentation,” OSS-Fuzz. (2025), [Online]. Available: <https://google.github.io/oss-fuzz/>.
- [71] Google, *Google/clusterfuzz*, Google, Apr. 9, 2025. [Online]. Available: <https://github.com/google/clusterfuzz>.
- [72] Google, *Google/fuzztest*, Google, Jul. 10, 2025. [Online]. Available: <https://github.com/google/fuzztest>.
- [73] Google, *Google/honggfuzz*, Google, Jul. 10, 2025. [Online]. Available: <https://github.com/google/honggfuzz>.
- [74] 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>.
- [75] Open Source Security Foundation (OpenSSF), *OSSF/fuzz-introspector*, Open Source Security Foundation (OpenSSF), Jun. 30, 2025. [Online]. Available: <https://github.com/ossf/fuzz-introspector>.
- [76] L. Thomason, *Leethomason/tinysql2*, Jul. 10, 2025. [Online]. Available: <https://github.com/leethomason/tinysql2>.
- [77] 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/>.
- [78] 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.
- [79] L. Torvalds, *Git*, Apr. 7, 2005. [Online]. Available: <https://git-scm.com/>.
- [80] 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.
- [81] D. A. Wheeler. “Flawfinder Home Page,” Flawfinder. (), [Online]. Available: <https://dwheeler.com/flawfinder/>.
- [82] 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.
- [83] 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.
- [84] H. Chase, *LangChain*, Oct. 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>.
- [85] J. Liu, *LlamaIndex*, Nov. 2022. DOI: 10.5281/zenodo.1234. [Online]. Available: https://github.com/jerryliu/llama_index.
- [86] A. Cedilnik, B. Hoffman, B. King, K. Martin, and A. Neundorf, *CMake - Upgrade Your Software Build System*, 2000. [Online]. Available: <https://cmake.org/>.
- [87] 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>.
- [88] E. Martin, *Ninja-build/ninja*, ninja-build, Jul. 14, 2025. [Online]. Available: <https://github.com/ninja-build/ninja>.
- [89] J. Pakkanen, *Mesonbuild/meson*, The Meson Build System, Jul. 14, 2025. [Online]. Available: <https://github.com/mesonbuild/meson>.
- [90] Google, *Google/atheris*, Google, Apr. 9, 2025. [Online]. Available: <https://github.com/google/atheris>.

1111 [91] Clibs Project. “Clib Packages,” GitHub. (2025), [Online]. Available: [https://github.com/clibs/clib/wiki/](https://github.com/clibs/clib/wiki/Packages)
1112 [Packages](https://github.com/clibs/clib/wiki/Packages).