


1 **OverHAuL: Harnessing Automation for C**
2 **Libraries with Large Language Models**

3 **BSc Thesis**

4 Konstantinos Chousos 
sdi2000215@di.uoa.gr

Department of Informatics and Telecommunications
National and Kapodistrian University of Athens

5 July, 2025

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

12 Preface

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

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

perseverance.

23 Acknowledgments

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

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

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

Table of contents

35	1. Introduction	1
36	1.1. Thesis Structure	2
37	1.2. Summary of Contributions	2
38	2. Background	3
39	2.1. Fuzz Testing	3
40	2.1.1. Motivation	5
41	2.1.2. Methodology	6
42	2.1.3. Challenges in Adoption	8
43	2.2. Large Language Models	8
44	2.2.1. State-of-the-art GPTs	9
45	2.2.2. Prompting	9
46	2.2.3. LLMs for Coding	10
47	2.2.4. LLMs for Fuzzing	11
48	2.3. Neurosymbolic AI	12
49	3. OverHAuL's Design	13
50	3.1. Installation and Usage	14
51	3.2. Architecture	15
52	3.2.1. Project Analysis	15
53	3.2.2. Harness Creation	16
54	3.2.3. Harness Evaluation	16
55	3.3. OverHAuL Techniques	17
56	3.3.1. Feedback Loop	17
57	3.3.2. React Agents Triplet	17
58	3.3.3. Codebase Oracle	18
59	3.4. High-Level Algorithm	19
60	3.5. Scope	20
61	3.6. Implementation	20
62	3.6.1. Development Tools	21
63	3.6.2. Reproducibility	21
64	4. Evaluation	23
65	4.1. Experimental Benchmark	23
66	4.1.1. Local Benchmarking	24

67	4.2. Results	25
68	4.2.1. RQ 1: Can OverHAuL generate working harnesses for unfuzzed C projects?	27
69	4.2.2. RQ2: What characteristics do these harnesses have? Are they similar to	
70	man-made harnesses?	27
71	4.2.3. RQ3: How do LLM usage patterns influence the generated harnesses?	27
72	4.2.4. RQ4: How do different symbolic techniques affect the generated harnesses?	28
73	4.3. Discussion	29
74	4.3.1. Threats to Validity	29
75	5. Related work	31
76	5.1. KLEE	31
77	5.2. IRIS	31
78	5.3. FUDGE	32
79	5.4. UTopia	32
80	5.5. FuzzGen	32
81	5.6. IntelliGen	33
82	5.7. CKGFuzzer	34
83	5.8. PromptFuzz	34
84	5.9. OSS-Fuzz	35
85	5.10. OSS-Fuzz-Gen	35
86	5.11. AutoGen	36
87	5.12. Differences	36
88	6. Future Work	39
89	6.1. Enhancements to Core Features	39
90	6.2. Experimentation with Large Language Models and Data Representation	40
91	6.3. Comprehensive Evaluation and Benchmarking	40
92	6.4. Practical Deployment and Community Engagement	41
93	7. Conclusion	42
94	Bibliography	43
95	Appendices	51
96	A. Abandoned Techniques	51
97	B. Sample Generated Harnesses	53
98	B.1. clibs/buffer	53
99	B.2. willem/cbuffer	55
100	B.3. dvhar/dateparse	58
101	B.4. h2non/semver.c	59
102	C. DSPy Custom Signatures	63

103 List of Figures

104	3.1. OverHAuL Workflow	13
105	3.2. OverHAuL execution example	15
106	4.1. Benchmark Results	25
107	4.2. Iterations Heatmap	26

108

List of Listings

109

2.1. Fuzzing harness format 6

110

2.2. Example fuzzing harness 7

111

2.3. Compilation of harness 7

112

3.1. OverHAuL installation 22

113

3.2. DSPy example 22

114

4.1. Sample dateparse harness 30

115 List of Tables

116	4.1. The benchmark project corpus. Each project name links to its corresponding	
117	GitHub repository. Each is followed by a short description and its GitHub stars	
118	count, as of July 18th, 2025.	24

1. Introduction

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

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

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

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

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

161 1.1. Thesis Structure

162 qqqqqqqq: Refactor when structure stabilizes

163 This thesis begins by outlining the foundational concepts necessary to understand its context
164 (Chapter 2) and progresses to a thorough survey of existing research in the field of automated
165 fuzzing (Chapter 5). We illustrate that the majority of contemporary fuzzing systems either de-
166 pend on pre-existing harnesses or utilize client code, frequently placing the burden of validation
167 and integration on the user. Next, we present the OverHAuL system, detailing its architecture
168 and the innovative techniques that underpin its implementation, as well as their contributions to
169 the advancement of automated harness generation (Chapter 3). Lastly, we compile a benchmark
170 dataset consisting of ten open-source C projects and rigorously assess OverHAuL’s performance
171 (Chapter 4, 4.2).

172 1.2. Summary of Contributions

173 This thesis presents the following key contributions:

- 174 1. The introduction of OverHAuL, a framework that enables fully automated end-to-end
175 fuzzing harness generation using LLMs. It introduces novel techniques like an iterative
176 feedback loop between LLM agents and the usage of a codebase oracle for code exploration.
- 177 2. Empirical validation through benchmarking experiments using ten real-world open source
178 projects. We demonstrate that OverHAuL successfully generates effective fuzzing har-
179 nesses with a chance of **81.25%**.
- 180 3. Full open sourcing of all research artifacts, datasets, and code at [https://github.com/](https://github.com/kchousos/OverHAuL)
181 [kchousos/OverHAuL](https://github.com/kchousos/OverHAuL) to encourage further research and ensure reproducibility.

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

2. Background

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

2.1. Fuzz Testing

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

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

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

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

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

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

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

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

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

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

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

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

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

240 **Definition 2.6** (Seed). An input given to the PUT that is mutated by the fuzzer to produce new
241 test cases. During a fuzz campaign (Definition 2.4) all seeds are stored in a seed *pool* or *corpus*
242 [13].

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

2.1.1. Motivation

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

— OWASP Foundation [21]

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

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

2.1.1.1. Success Stories

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

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

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

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

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

284 2.1.2. Methodology

285 As previously discussed, fuzz testing of a PUT is typically conducted using a dedicated fuzzing
286 engine (Definition 2.3). Among the most widely adopted fuzzers for C and C++ projects and
287 libraries are AFL [27]—which has since evolved into AFL++ [30]—and LibFuzzer [19]. Within the
288 OverHAuL framework, LibFuzzer is preferred due to its superior suitability for library fuzzing,
289 whereas AFL++ predominantly targets executables and binary fuzzing.

290 2.1.2.1. LibFuzzer

291 LibFuzzer [19] is an in-process, coverage-guided evolutionary fuzzing engine primarily designed
292 for testing libraries. It forms part of the LLVM ecosystem [31] and operates by linking directly
293 with the library under evaluation. The fuzzer delivers mutated input data to the library through
294 a designated fuzzing entry point, commonly referred to as the *fuzz target* or *harness*.

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

298 For the remainder of this thesis, the terms presented in Definition 2.7 will be used interchange-
299 ably.

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

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

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

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

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

306 To compile and link such a harness with LibFuzzer, the Clang compiler—also part of the LLVM
307 project [31]—must be used alongside appropriate compiler flags. For instance, compiling the
308 harness in Listing 2.2 can be achieved as shown in Listing 2.3.

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

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

309 2.1.2.2. AFL and AFL++

310 *American Fuzzy Lop* (AFL) [27], developed by Michał Zalewski, is a seminal fuzzer targeting C
311 and C++ applications. Its core methodology relies on instrumented binaries to provide edge
312 coverage feedback, thereby guiding input mutation towards unexplored program paths. AFL
313 supports several emulation backends including QEMU [32]—an open-source CPU emulator
314 facilitating fuzzing on diverse architectures—and Unicorn [33], a lightweight multi-platform CPU
315 emulator. While AFL established itself as a foundational tool within the fuzzing community, its
316 successor AFL++ [30] incorporates numerous enhancements and additional features to improve
317 fuzzing efficacy.

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

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

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

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

331 2.1.3. Challenges in Adoption

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

343 2.2. Large Language Models

344 Natural Language Processing (NLP), a subfield of AI, has a rich and ongoing history that has
345 evolved significantly since its beginning in the 1990s [34], [35]. Among the most notable—and
346 recent—advancements in this domain are LLMs, which have transformed the landscape of NLP
347 and AI in general.

348 At the core of many LLMs is the attention mechanism, which was introduced by Bahdanau
349 et al. in 2014 [36]. This pivotal innovation enabled models to focus on relevant parts of the
350 input sequence when making predictions, significantly improving language understanding and
351 generation tasks. Building on this foundation, the Transformer architecture was proposed by
352 Vaswani et al. in 2017 [37]. This architecture has become the backbone of most contemporary
353 LLMs, as it efficiently processes sequences of data, capturing long-range dependencies without
354 being hindered by sequential processing limitations.

One of the first major breakthroughs utilizing the Transformer architecture was BERT (Bidirectional Encoder Representations from Transformers), developed by Devlin et al. in 2019 [38]. BERT’s bi-directional understanding allowed it to capture the context of words from both directions, which improved the accuracy of various NLP tasks. Following this, the Generative Pre-trained Transformer (GPT) series, initiated by OpenAI with the original GPT model in 2018 [39], further pushed the boundaries. Subsequent iterations, including GPT-2 [40], GPT-3 [41], and the most current GPT-4 [42], have continued to enhance performance by scaling model size, data, and training techniques.

In addition to OpenAI’s contributions, other significant models have emerged, such as Claude, DeepSeek-R1 and the Llama series (1 through 3) [43]–[45]. The proliferation of LLMs has sparked an active discourse about their capabilities, applications, and implications in various fields.

2.2.1. State-of-the-art GPTs

User-facing LLMs are generally categorized between closed-source and open-source models. Closed-source LLMs like ChatGPT, Claude, and Gemini [43], [46], [47] 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 [45] and Deepseek [44], provide researchers and practitioners with access to model weights, allowing for greater transparency and adaptability.

2.2.2. Prompting

Interaction with LLMs typically occurs through chat-like interfaces where the user gives queries and tasks for the LLM to answer and complete, a process commonly referred to as *prompting*. A critical aspect of effective engagement with LLMs is the usage of different prompting strategies, which can significantly influence the quality and relevance of the generated outputs. Various approaches to prompting have been developed and studied, including zero-shot and few-shot prompting. In zero-shot prompting, the model is expected to perform the given task without any provided examples, while in few-shot prompting, the user offers a limited number of examples to guide the model’s responses [41].

To enhance performance on more complex tasks, several advanced prompting techniques have emerged. One notable strategy is the *Chain of Thought* approach (COT) [48], which entails presenting the model with sample thought processes for solving a given task. This method encourages the model to generate more coherent and logical reasoning by mimicking human-like cognitive pathways. A more refined but complex variant of this approach is the *Tree*

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

of *Thoughts* technique [49], which enables the LLM to explore multiple lines of reasoning concurrently, thereby facilitating the selection of the most promising train of thought for further exploration.

In addition to these cognitive strategies, Retrieval-Augmented Generation (RAG) [50] is another innovative technique that enhances the model’s capacity to provide accurate information by incorporating external knowledge not present in its training dataset. RAG operates by integrating the LLM with an external storage system—often a vector store containing relevant documents—that the model can query in real-time. This allows the LLM to pull up pertinent and/or proprietary information in response to user queries, resulting in more comprehensive and accurate answers.

Moreover, the ReAct framework [51], which stands for Reasoning and Acting, empowers LLMs by granting access to external tools. This capability allows LLM instances to function as intelligent agents that can interact meaningfully with their environment through user-defined functions. For instance, a ReAct tool could be a function that returns a weather forecast based on the user’s current location. In this scenario, the LLM can provide accurate and truthful predictions, thereby mitigating risks associated with hallucinated responses.

2.2.3. LLMs for Coding

The impact of LLMs in software development in recent years is apparent, with hundreds of LLM-assistance extensions and Integrated Development Environments (IDEs) being published. Notable instances include tools like GitHub Copilot and IDEs such as Cursor [52], [53], which leverage LLM capabilities to provide developers with coding suggestions, auto-completions, and even real-time debugging assistance. Such innovations have introduced a layer of interaction that enhances productivity and fosters a more intuitive coding experience. Additionally, more and more LLMs are now specifically trained for usage in code-generation tasks [54]–[56].

One exemplary product of this innovation is *vibecoding* and the no-code movement, which describe the development of software by only prompting and tasking an LLM, i.e. without any actual programming required by the user. This constitutes a showcase of how LLMs can be used to elevate the coding experience by supporting developers as they navigate complex programming tasks [57]. By analyzing the context of the code being written, these sophisticated models can provide contextualized insights and relevant snippets, effectively streamlining the development process. Developers can benefit from reduced cognitive load, as they receive suggestions that not only cater to immediate coding needs but also promote adherence to best practices and coding standards.

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

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

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

436 **2.2.4. LLMs for Fuzzing**

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

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

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

2.3. Neurosymbolic AI

Neurosymbolic AI represents a groundbreaking fusion of neural network methodologies with symbolic execution techniques and tools, providing a multi-faceted approach to overcoming the inherent limitations of traditional AI paradigms [58], [59]. This innovative synthesis seeks to combine the strengths of both neural networks, which excel in pattern recognition and data-driven learning, and symbolic systems, which offer structured reasoning and interpretability. By integrating these two approaches, neurosymbolic AI aims to create cognitive models that are not only more accurate but also more robust in problem-solving contexts.

At its core, Neurosymbolic AI facilitates the development of AI systems that are capable of understanding and interpreting feedback in real-world scenarios [60]. This characteristic is particularly significant in the current landscape of artificial intelligence, where LLMs are predominant. In this context, Neurosymbolic AI is increasingly viewed as a critical solution to pressing issues related to explainability, attribution, and reliability in AI systems [61], [62]. These challenges are essential for ensuring that AI systems can be trusted and effectively utilized in various applications, from business to healthcare.

The burgeoning field of neurosymbolic AI is still in its nascent stages, with ongoing research and development actively exploring its potential to enhance attribution methodologies within large language models. By addressing these critical challenges, Neurosymbolic AI can significantly contribute to the broader landscape of trustworthy AI systems, allowing for more transparent and accountable decision-making processes [58], [61], [62].

Moreover, the application of neurosymbolic AI within the domain of fuzzing is gaining traction, paving the way for innovative explorations. This integration of LLMs with symbolic systems opens up new avenues for research. Currently, there are only a limited number of tools that support such hybrid approaches (Chapter 5). Among these, OverHAuL constitutes a Neuro[Symbolic] tool, as classified by Henry Kautz’s taxonomy [63], [64]. This means that the neural model—specifically the LLM—can leverage symbolic reasoning tools—in this case a source code explorer (Section 3.6)—to augment its reasoning capabilities. This symbiotic relationship enhances the overall efficacy and versatility of LLMs for fuzzing harnesses generation, demonstrating the profound potential held by the fusion of neural and symbolic methodologies.

3. OverHAuL’s Design

In this thesis we present **OverHAuL** (**H**arness **A**utomation with **L**LMs), a neurosymbolic AI tool that automatically generates fuzzing harnesses for C libraries through LLM agents. In its core, OverHAuL is comprised by three LLM ReAct agents [51]—each with its own responsibility and scope—and a vector store index reserving the given project’s analyzed codebase. An overview of OverHAuL’s process is presented in Figure 3.1. The objective of OverHAuL is to streamline the process of fuzz testing for C libraries. Given a link to a git repository [65] of a C library, OverHAuL automatically generates a new fuzzing harness specifically designed for the project. In addition to the harness, it produces a compilation script to facilitate building the harness, generates a representative input that can trigger crashes, and logs the output from the executed harness.

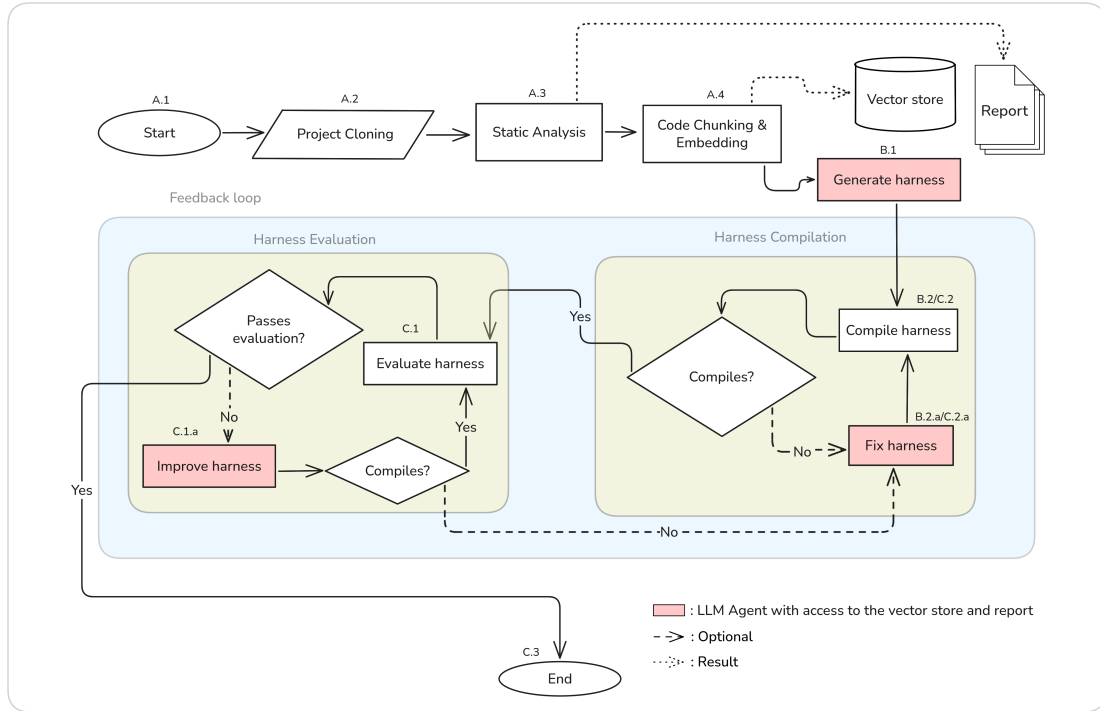


Figure 3.1: Overview of OverHAuL’s automatic harnessing process.

As detailed in Section 5.12, OverHAuL does not expect and depend on the existence of client code or unit tests [10]–[12] nor does it require any preexisting fuzzing harnesses [9] or any documentation present [66]. Also importantly, OverHAuL is decoupled from other fuzzing

504 projects, thus lowering the barrier to entry for new projects [9], [67]. Lastly, the user isn't
505 mandated to manually specify the function which the harness-to-be-generated must fuzz.
506 Instead, OverHAuL's agents examine and assess the provided codebase, choosing after evaluation
507 the most optimal target function.

508 OverHAuL utilizes autonomous ReAct agents which inspect and analyze the project's source
509 code. The latter is stored and interacted with as a set of text embeddings [68], kept in a vector
510 store. Both approaches are, to the best of our knowledge, novel in the field of automatic fuzzing
511 harnesses generation. OverHAuL also implements an evaluation component that assesses in
512 real-time all generated harnesses, making the results tenable, reproducible and well-founded.
513 Ideally, this methodology provides a comprehensive and systematic framework for identifying
514 previously unknown software vulnerabilities in projects that have not yet been fuzz tested.

515 Finally, OverHAuL excels in its user-friendliness, as it constitutes a simple and easily-installable
516 Python package with minimal external dependencies—only real dependency being Clang, a
517 prevalent compiler available across all primary operating systems. This contrasts most other
518 comparable systems, which are typically characterized by their limited documentation, lack
519 of extensive testing, and a focus primarily on experimental functionality. For instance, both
520 fuzz-introspector and OSS-Fuzz-Gen are specifically designed for integration with the OSS-Fuzz
521 platform [9], [67], [69]. When utilized outside this environment, they require users to operate
522 directly from the project's root directory and interact with the tools primarily through unrefined
523 Python scripts, thereby limiting their accessibility and ease of use.

524 3.1. Installation and Usage

525 The source code of OverHAuL is available in <https://github.com/kchousos/OverHAuL>. Over-
526 HAuL can be installed by cloning the git repository locally, creating and enabling a Python3.10
527 virtual environment [70] and installing it inside the environment using Python's PIP package
528 installer [71], like in Listing 3.1.

529 To use OverHAuL, you need to provide a secret key for using OpenAI's API service. This key
530 can be either stored in a .env file in the root directory, like so:

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

531 Or it can be exported in the shell environment:

```
1 $ export OPENAI_API_KEY=<API-key-here>
2 $ overhaul <repo-link>
```

532 Once these preliminary steps are completed, OverHAuL can be executed. The primary argument
533 required by OverHAuL is the repository link of the library that is to be fuzzed. Additionally,

users have the option to specify certain command-line flags, which allow them to control the checked-out commit of the cloned project, select the OpenAI LLM model from a predefined list, define specific file patterns for OverHAuL to search for, and determine the directory in which the project will be cloned. A sample successful execution can be presented in Figure 3.2.

```

> overhaul https://github.com/dvhar/dateparse -m gpt-4.1
2025-07-17 00:55:18.087 | INFO | overhaul.cli:parse_arguments:184 - Cloning project's repo in the output/dateparse directory...
2025-07-17 00:55:18.976 | INFO | overhaul.core:harnesser:_init_:167 - Initializing LLM...
2025-07-17 00:55:18.991 | INFO | overhaul.cli:main:243 - Iteration 1 of harnessing...
2025-07-17 00:55:18.991 | INFO | overhaul.core:harnesser:harness: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-dfaa34d0e98889cd82d2cd680cf96fd04552a2b4', 1752782962.4113252))
2025-07-17 00:56:02.419 | SUCCESS | overhaul.cli:main:282 - All done!

```

Figure 3.2.: A successful execution of OverHAuL, harnessing [dvhar's dateparsing C library](#), using OpenAI's gpt-4.1 model. Debug statements are printed to showcase the interaction between the LLM agents and the codebase oracle (Section 3.3.3).

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

3.2. Architecture

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

3.2.1. Project Analysis

In the project analysis stage (steps A.1–A.4), the project to be fuzzed is ran through a static analysis tool named Flawfinder [72] 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. Flawfinder 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 LLMs.

3.2.2. Harness Creation

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

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

3.2.3. Harness Evaluation

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

1. The harness has no memory leaks during its execution
This is inferred by the existence of `leak-<hash>` files.
2. A new testcase was created *or* the harness executed for at least `MIN_EXECUTION_TIME` (i.e. did not crash on its own)
When a crash happens, and thus a testcase is created, it results in a `crash-<hash>` file.

594 3. The created testcase is not empty

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

596 Similarly to the second stage’s compilation phase (steps B.2–B.2.a), if a harness does not pass
597 the evaluation for whatever reason it is sent to an “improver” agent. This agent is instructed
598 to refine it based on its code and cause of failing the evaluation. This process is also iterative.
599 If any of the improved harness versions fail to compile, the aforementioned “fixer” agent is
600 utilized again (steps C.2–C.2.a). All produced crash files and the harness execution output are
601 saved in the project’s directory.

602 3.3. OverHAuL Techniques

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

607 3.3.1. Feedback Loop

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

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

620 3.3.2. React Agents Triplet

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

625 By assigning individual contexts to the agents, we enable a broader exploration of possibilities
626 during each run. For instance, the “improver” agent can investigate alternative pathways or

strategies that the “generator” agent may have potentially overlooked or internally deemed inadequate inaccurately. This separation not only fosters a more diverse range of solutions but also enhances the overall robustness of the system by allowing for iterative refinement based on each agent’s unique insights.

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

3.3.3. Codebase Oracle

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

By structuring the codebase into chunks at the level of individual functions, LLM agents can engage with the code more effectively by focusing on its functional components. This methodology not only allows for a more nuanced understanding of the codebase but also ensures that the responses generated do not consume an excessive portion of the limited context window available to the agents. In contrast, if the codebase were organized and queried at the file level, the chunks of information would inevitably become larger, leading to an increase in noise and a dilution of meaningful content in each chunk [73]. Given the constant size of the embeddings used in processing, each progressively larger chunk would be less semantically significant, ultimately compromising the quality of the retrieval process.

Defining the function as the primary unit of analysis represents the most proportionate balance between the size of the code segments and their semantic significance. It serves as the ideal “zoom-in” level for the exploration of code, allowing for greater clarity and precision in understanding the functionality of individual code segments. This same principle is widely recognized in the training of code-specific LLMs, where a function-level approach has been shown to enhance performance and comprehension [74]. By adopting this methodology, we aim to foster a more robust interaction between LLM agents and the underlying codebase, ultimately facilitating a more effective and efficient exploration process.

3.4. High-Level Algorithm

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

Algorithm 3.1 OverHAuL

Require: *repository*

Ensure: *harness, compilation_script, crash_input, execution_log*

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

3.5. Scope

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

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

Lastly, No support for build systems such as Make or CMake [75], [76] is yet implemented. Such functionality would exponentially increase the complexity of the build step and is beyond the scope of this thesis.

3.6. Implementation

In creating the codebase oracle, we employ the “libclang” Python package [77] to slice functions based on the AST capability provided by Clang. As detailed in Section 3.3.3, the intermediate output consists of a list of Python dictionaries, with each dictionary storing a function’s body, signature, and corresponding file path. Each chunk of function code is then converted into an embedding using OpenAI’s “text-embedding-3-small” model [78] and stored in a FAISS vector store index [79]. This index is mapped to a metadata structure that contains the aforementioned function data—specifically the actual function body, signature, and file path. When a search is conducted on the index, the results returned are the embeddings. The responses that the LLM agent receives are derived from the corresponding metadata entries of each embedding.

All LLM agents and components are developed using the DSPy library, a declarative Python framework for LLM programming created by Stanford’s NLP research team [80]. DSPy offers built-in modules and abstractions that facilitate the composition of LLMs and prompting techniques, such as Chain of Thought and ReAct (Listing 3.2). Each agent within OverHAuL is an instance of DSPy’s ReAct module [81], accompanied by a custom Signature [82]—displayed in Appendix C. DSPy was selected over other contemporary LLM libraries, such as LangChain and Llamaindex [83], [84], because of its user-friendliness, logical abstractions, and efficient development process—qualities that are often lacking in these alternative libraries [85]–[87].

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

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

709 3.6.1. Development Tools

710 The development of OverHAuL incorporates a variety of tools aimed at enhancing functionality
711 and efficiency. Notably, “uv” is a Python package and project manager written in Rust that serves
712 as a replacement for Poetry. Additionally, “Ruff,” a code linter and formatter also developed in
713 Rust, contributes to code quality by enforcing consistent formatting standards. The project also
714 employs “MyPy,” the widely-used static type checker for Python, to ensure type correctness.
715 Testing is facilitated through “PyTest,” a robust Python testing framework. Lastly, “pdoc” is
716 utilized as a Static Site Generator (SSG) to automate the creation of API documentation¹ [88]–
717 [92].

718 3.6.2. Reproducibility

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

¹Available at <https://kchousos.github.io/OverHAuL/>.

Listing 3.1 OverHAuL's installation process.

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

Listing 3.2 Sample DSPy program.

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

4. Evaluation

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

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

4.1. Experimental Benchmark

To evaluate OverHAuL, a benchmarking script was implemented¹ and a corpus of ten open-source C libraries was assembled. This collection comprises firstly of user dhvar’s “dateparse” library, which is also used as a running example in OSS-Fuzz-Gen’s [9] experimental from-scratch harnessing feature (Section 5.10). Secondly, nine other libraries chosen randomly² from the package catalog of Clib, a “package manager for the C programming language” [93], [94]. All libraries can be seen Table 4.1, along with their descriptions.

OverHAuL was evaluated through the experimental benchmark from 6th of June, 2025 to 18th of July, 2025, using OpenAI’s gpt-4.1-mini model [95]. For these runs, each OverHAuL execution was configured with a 5 minute harness execution timeout and an iteration budget of 10. Each benchmark run was executed as a GitHub Actions workflow on Linux virtual machines with 4-vCPUs and 16GiB of memory hosted on Microsoft Azure [96], [97]. The result directory (as described in Section 4.1.1) for each is available as a downloadable artifact in the corresponding GitHub Actions entry.

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

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

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

Project	Description	Stars	SLOC
dvhar/dateparse	A library that allows parsing dates without knowing the format in advance.	2	2272
clibs/buffer	A string manipulation library.	204	354
jwerle/libbeaufort	A library implementation of the Beaufort cipher [98].	13	321
jwerle/libbacon	A library implementation of the Baconian cipher [99].	8	191
jwerle/chfreq.c	A library for computing the character frequency in a string.	5	55
jwerle/progress.c	A library for displaying progress bars in the terminal.	76	357
willemt/cbuffer	A circular buffer implementation.	261	170
willemt/torrent-reader	A torrent-file reader library.	6	294
orangeduck/mpc	A type-generic parser combinator library.	2,753	3632
h2non/semver.c	A semantic version v2.0 parsing and rendering library [100].	190	608

4.1.1. Local Benchmarking

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

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

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

4.2. Results

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

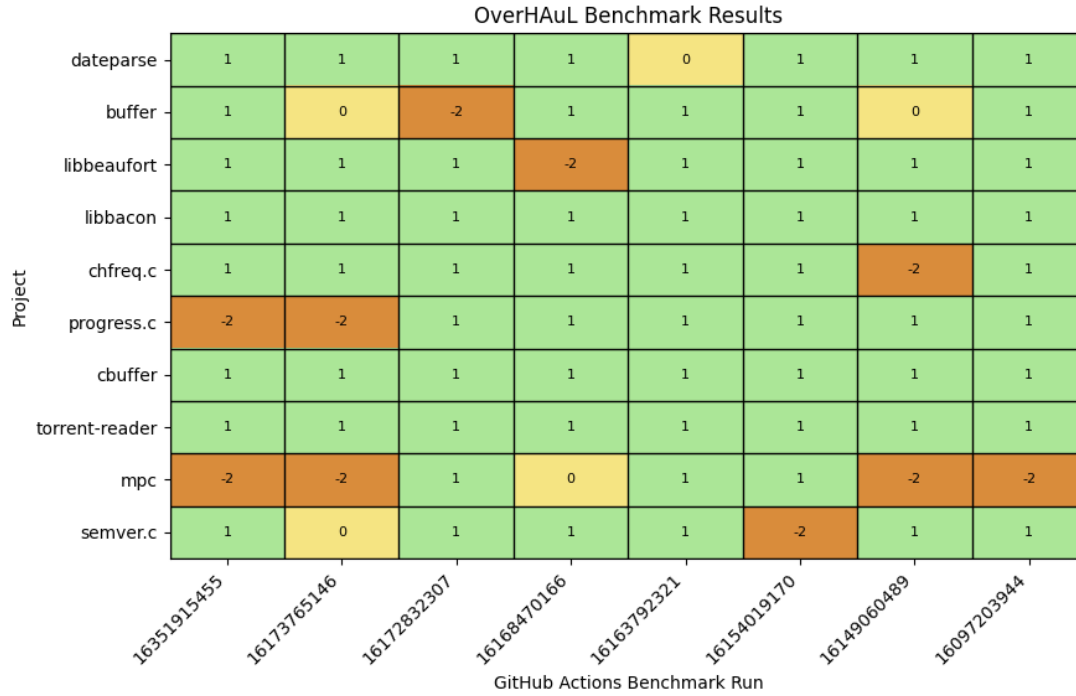


Figure 4.1.: The benchmark results for OverHAuL are illustrated with the y-axis depicting the ten-project corpus outlined in Section 4.1. The x-axis represents the various benchmark runs. Each label constitutes a unique hash identifier corresponding to a specific GitHub Actions workflow run, which can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

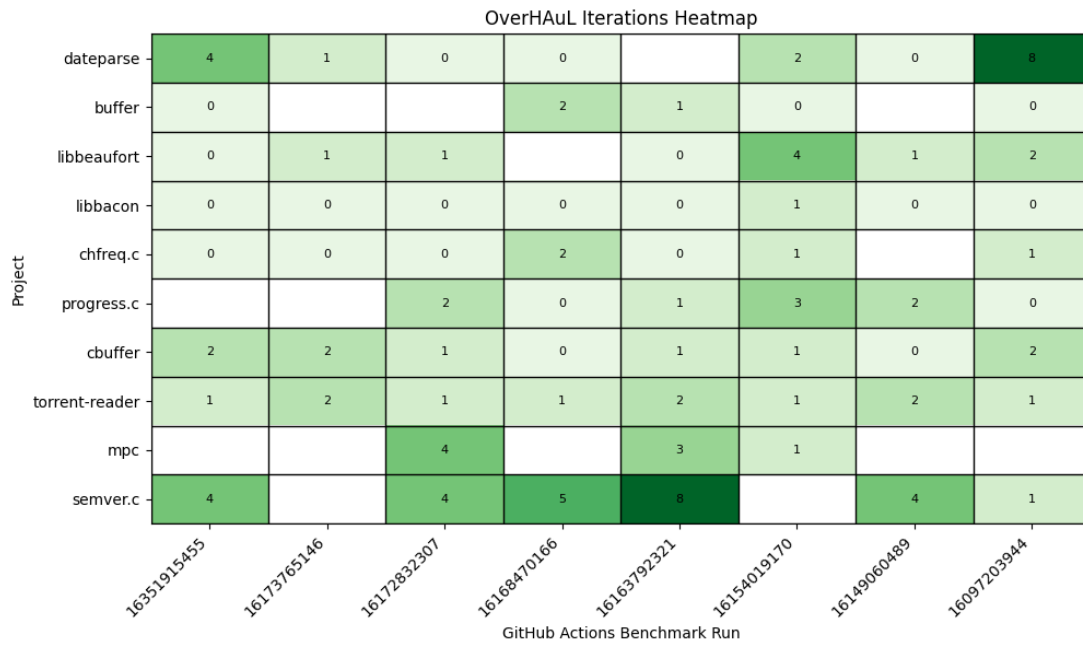


Figure 4.2.: This heatmap illustrates the number of iterations required for each project to be successfully harnessed, as determined by the benchmark results. Higher color intensity corresponds to a greater number of iterations needed for successful harnessing. Cells left blank signify instances where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

770 **4.2.1. RQ 1: Can OverHAuL generate working harnesses for unfuzzed C**
771 **projects?**

772 OverHAuL demonstrates a notably high success rate in generating fuzzing harnesses that
773 effectively uncover crash-inducing inputs for target programs, achieving a rate of **81.25%**.
774 Furthermore, while the collected data indicates that OverHAuL has not produced any non-
775 compilable harnesses, it is prudent to acknowledge the possibility—however unlikely—such
776 occurrences. Taken together, these findings provide a robust and affirmative answer to RQ1.

777 **4.2.2. RQ2: What characteristics do these harnesses have? Are they similar to**
778 **man-made harnesses?**

779 From sampling OverHAuL’s generated harnesses, the answer to RQ2 remains unclear. Most
780 of the time, the fuzz targets that are produced are understandable and similar to something
781 a software engineer might program. Take for example Listing 4.1. Nonetheless, sometimes
782 generated harnesses contain usage of inexplicable or arbitrary constants and peculiar control
783 flow checks. This makes them harder to understand and quite possibly incorrect in many
784 cases, thus diverging from seeming human-written. Additionally, the generated harnesses differ
785 significantly across projects and executions, both in size and complexity (Appendix B). RQ2’s
786 answer remains an unclear “it depends”, given the variance in OverHAuL’s results.

787 **4.2.3. RQ3: How do LLM usage patterns influence the generated harnesses?**

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

791 All benchmark experiments on GitHub’s infrastructure were conducted using OpenAI’s gpt-
792 4.1-mini. Preliminary local testing included a spectrum of models—gpt-4.1, gpt-4o, gpt-4, and
793 gpt-3.5-turbo. Notably, both gpt-4.1 and gpt-4.1-mini achieved comparable performance,
794 consistently generating robust fuzzing harnesses. In contrast, gpt-4o yielded somewhat average
795 results, while gpt-4 and gpt-3.5-turbo exhibited significantly inferior performance, averaging
796 only 2 out of 10 projects successfully harnessed per benchmark run. Models with suboptimal
797 performance were excluded in subsequent development phases. These findings underscore
798 the necessity of selecting advanced LLM architectures to realize OverHAuL’s potential; in
799 particular, gpt-4o represents a recent baseline for acceptable performance. Because LLM
800 model capabilities are evolving rapidly, it is reasonable to anticipate ongoing improvements in
801 OverHAuL’s harness-generation efficacy as newer LLMs become available.

802 Prompting methodology is equally crucial. The adoption of ReAct prompting has proven most
803 effective in the current implementation of OverHAuL [51]. Alternative prompting paradigms—
804 including zero-shot and Chain-of-Thought (COT) approaches [48]—were empirically evaluated,
805 as detailed in Appendix A, but failed to deliver satisfactory outcomes. A central challenge

in automated harness generation involves ensuring that the resulting harness is both compilable and operationally effective. This alignment with real-world constraints necessitates continuous interaction between the LLM and the target environment, best achieved through agentic workflows [101]. The superior performance of ReAct prompting likely stems from its structured approach to iterative code exploration and refinement, facilitating a cycle of observation, planning, and action that is particularly well-suited to harness synthesis.

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

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

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

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

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

4.3. Discussion

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

A noteworthy consideration in our benchmarking setup is the possibility that some of the open-source libraries evaluated may have been included in the LLM’s training data. This introduces a risk of overestimating OverHAuL’s performance on code that is unseen or proprietary. Results for closed-source or less widely available libraries could therefore be weaker. Nonetheless, this potential limitation can theoretically be addressed through targeted fine-tuning of the LLM [102], [103].

4.3.1. Threats to Validity

Our evaluation of OverHAuL was conducted on ten relatively obscure open-source C libraries representing a range of application domains and functionalities. While this selection reduces the likelihood that these projects were used in LLM training and thus minimizes potential bias, it remains uncertain how transferable our results are to larger, more complex, or structurally different codebases. Factors such as varying design paradigms, architectural patterns, or real-world deployment contexts may pose new challenges for OverHAuL’s scalability and effectiveness.

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

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

Listing 4.1 Sample harness for dateparse, generated by OverHAuL.

```
1  #include <stddef.h>
2  #include <stdint.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include "dateparse.h"
6
7  // No artificial size limit to allow fuzzer to explore full
8  // input size for boundaries
9
10 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
11     // Allocate buffer with extra byte for null termination
12     char *input_str = (char *)malloc(size + 1);
13     if (!input_str) {
14         return 0;
15     }
16
17     memcpy(input_str, data, size);
18     input_str[size] = '\0';
19
20     date_t parsed_date = 0;
21     int offset = 0;
22
23     // Array of string lengths targeting boundary
24     // (including 0 = internal strlen)
25     size_t test_lens[] = {0, size, size > 0 ? size - 1 : 0, 12, 13, 14};
26
27     for (size_t i = 0; i < sizeof(test_lens) / sizeof(test_lens[0]); i++) {
28         size_t len = test_lens[i];
29         if (len ≤ size) {
30             dateparse(input_str, &parsed_date, &offset, (int)len);
31         }
32     }
33
34     free(input_str);
35     return 0;
36 }
```

5. Related work

Automated testing, automated fuzzing and automated harness creation have a long research history. Still, a lot of ground remains to be covered until true automation of these tasks is achieved. Until the introduction of transformers [37] and the 2020’s boom of commercial GPTs [46], automation regarding testing and fuzzing was mainly attempted through static and dynamic program analysis methods. These approaches are still utilized, but the fuzzing community has shifted almost entirely to researching the incorporation and employment of LLMs in the last half decade [9]–[12], [66], [104]–[108]. The following works stand out as the most notable in the field.

5.1. KLEE

KLEE [109] is a seminal and widely cited symbolic execution engine introduced in 2008 by Cadar et al. It was designed to automatically generate high-coverage test cases for programs written in C, using symbolic execution to systematically explore the control flow of a program. KLEE operates on the LLVM [31] bytecode representation of programs, allowing it to be applied to a wide range of C programs compiled to the LLVM intermediate representation.

Instead of executing a program on concrete inputs, KLEE performs symbolic execution—that is, it runs the program on symbolic inputs, which represent all possible values simultaneously. At each conditional branch, KLEE explores both paths by forking the execution and accumulating path constraints (i.e., logical conditions on input variables) along each path. This enables it to traverse many feasible execution paths in the program, including corner cases that may be difficult to reach through random testing or manual test creation.

When an execution path reaches a terminal state (e.g., a program exit, an assertion failure, or a segmentation fault), KLEE uses a constraint solver to compute concrete input values that satisfy the accumulated constraints for that path. These values form a test case that will deterministically drive the program down that specific path when executed concretely.

5.2. IRIS

IRIS [104] 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

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

905 5.3. FUDGE

906 FUDGE [12] is a closed-source tool, made by Google, for automatic harness generation of
907 C and C++ projects based on existing client code. It was used in conjunction with and in
908 the improvement of Google’s OSS-Fuzz [67] (Section 5.9). Being deployed inside Google’s
909 infrastructure, FUDGE continuously examines Google’s internal code repository, searching
910 for code that uses external libraries in a meaningful and “fuzzable” way (i.e. predominantly
911 for parsing). If found, such code is *sliced* [110] based on its Abstract Syntax Tree (AST) using
912 LLVM’s Clang tool [31]. The above process results in a set of abstracted mostly-self-contained
913 code snippets that make use of a library’s calls and/or API. These snippets are later *synthesized*
914 into the body of a fuzz driver, with variables being replaced and the fuzz input being utilized.
915 Each is then injected in an LLVMFuzzerTestOneInput function and finalized as a fuzzing harness.
916 A building and evaluation phase follows for each harness, where they are executed and examined.
917 Every passing harness along with its evaluation results is stored in FUDGE’s database, reachable
918 to the user through a custom web-based UI.

919 5.4. UTopia

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

929 5.5. FuzzGen

930 Another project of Google is FuzzGen [11], this time open-source. Like FUDGE, it leverages
931 existing client code of the target library to create fuzz targets for it. FuzzGen uses whole-system
932 analysis, through which it creates an *Abstract API Dependence Graph* (A²DG). It uses the latter
933 to automatically generate LibFuzzer-compatible harnesses. For FuzzGen to work, the user needs
934 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. *A²DG construction mechanism*. For all the existing API calls, FuzzGen builds an A²DG to record the API usages and infers its intended structure. After completion, this directed graph contains all the valid API call sequences found in the client code corpus. It is built in a two-step process: First, many smaller A²DGs are created, one for each root function per client code snippet. Once such graphs have been created for all the available client code instances, they are combined to formulate the master A²DG. This graph can be seen as a template for correct usage of the library. 3. *Fuzzer generator*. Through the A²DG, a fuzzing harness is created. Contrary to FUDGE, FuzzGen does not create multiple “simple” harnesses but a single complex one with the goal of covering the whole A²DG. In other words, while FUDGE fuzzes a single API call at a time, FuzzGen’s result is a single harness that tries to fuzz the given library all at once through complex API usage.

5.6. IntelliGen

IntelliGen [111] is a system for automatically synthesizing fuzz drivers by statically identifying potentially vulnerable entry-point functions within C projects. Implemented using LLVM [31], IntelliGen focuses on Improving fuzzing efficiency by targeting code more likely to contain memory safety issues, rather than exhaustively fuzzing all available functions.

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

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

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

5.7. CKGFuzzer

CKGFuzzer [112] is a fuzzing framework designed to automate the generation of effective fuzz drivers for C/C++ libraries by leveraging static analysis and large language models. Its workflow begins by parsing the target project along with any associated library APIs to construct a code knowledge graph. This involves two primary steps: first, parsing the AST, and second, performing inter-procedural program analysis. Through this process, CKGFuzzer extracts essential program elements such as data structures, function signatures, function implementations, and call relationships.

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

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

Finally, any crashes triggered during fuzzing are subjected to a reasoning process based on chain-of-thought prompting (Section 2.2.2). To help determine their severity and root cause, CKGFuzzer consults an LLM-generated knowledge base containing real-world examples of vulnerabilities mapped to known CWE entries.

5.8. PromptFuzz

PromptFuzz [113] constitutes a framework for automatically generating fuzz drivers using LLMs, with a novel focus on *prompt mutation* to improve coverage. Its aim is to explore more of the API surface with each prompt iteration. It is implemented in Rust and targets C libraries.

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

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

1009 Once an effective program is produced, it is transformed into a fuzz driver by replacing constants
1010 and arguments with variables derived from the fuzzer input. Multiple such drivers are embedded
1011 into a single harness, where the input determines which program variant to execute, typically
1012 via a case-switch construct.

1013 5.9. OSS-Fuzz

1014 OSS-Fuzz [67], [114] is a continuous, scalable and distributed cloud fuzzing solution for critical
1015 and prominent open-source projects. Developers of such software can submit their projects
1016 to OSS-Fuzz’s platform, where its harnesses are built and constantly executed. This results in
1017 multiple bug findings that are later disclosed to the primary developers and are later patched.

1018 OSS-Fuzz started operating in 2016, an initiative in response to the Heartbleed vulnerability
1019 [22], [23], [25]. Its hope is that through more extensive fuzzing such errors could be caught
1020 and corrected before having the chance to be exploited and thus disrupt the public digital
1021 infrastructure. So far, it has helped uncover over 10,000 security vulnerabilities and 36,000
1022 bugs across more than 1,000 projects, significantly enhancing the quality and security of major
1023 software like Chrome, OpenSSL, and Systemd.

1024 A project that’s part of OSS-Fuzz must have been configured as a ClusterFuzz [115] project.
1025 ClusterFuzz is the fuzzing infrastructure that OSS-Fuzz uses under the hood and depends on
1026 Google Cloud Platform services, although it is possible to host it locally. Such an integration
1027 requires setting up a build pipeline, fuzzing jobs and expects a Google Developer account. Results
1028 are accessible through a web interface. ClusterFuzz, and by extension OSS-Fuzz, supports fuzzing
1029 through LibFuzzer, AFL++, Honggfuzz and FuzzTest—successor to Centipede— with the last two
1030 being Google projects [19], [30], [116], [117]. C, C++, Rust, Go, Python and Java/JVM projects
1031 are supported.

1032 5.10. OSS-Fuzz-Gen

1033 OSS-Fuzz-Gen (OFG) [9], [118] is Google’s current state-of-the-art project regarding automatic
1034 harness generation through LLMs. It’s purpose is to improve the fuzzing infrastructure of open-
1035 source projects that are already integrated into OSS-Fuzz. Given such a project, OSS-Fuzz-Gen
1036 uses its preexisting fuzzing harnesses and modifies them to produce new ones. Its architecture
1037 can be described as follows:

- 1038 1. With an OSS-Fuzz project’s GitHub repository link, OSS-Fuzz-Gen iterates through a
1039 set of predefined build templates and generates potential build scripts for the project’s
1040 harnesses.
- 1041 2. If any of them succeed they are once again compiled, this time through fuzz-introspector
1042 [69]. The latter constitutes a static analysis tool, with fuzzer developers specifically in
1043 mind.

- 1044 3. Build results, old harness and fuzz-introspector report are included in a template-generated
1045 prompt, through which an LLM is called to generate a new harness.
1046 4. The newly generated fuzz target is compiled and if it is done so successfully it begins
1047 execution inside OSS-Fuzz’s infrastructure.

1048 This method proves to be meaningful, with code coverage in fuzz campaigns increasing thanks
1049 to the new generated fuzz drivers. In the case of the tinysql2 project [119], line coverage went
1050 from 38% to 69% without any manual interventions [118].

1051 In 2024, OSS-Fuzz-Gen introduced an experimental feature for generating harnesses in previ-
1052 ously unfuzzed projects [120]. The code for this feature resides in the `experimental/from_scratch`
1053 directory of the project’s GitHub repository [9], with the latest known working commit being
1054 171aac2 and the latest overall commit being four months ago.

1055 5.11. AutoGen

1056 AutoGen [66] is a closed-source tool that generates new fuzzing harnesses, given only the
1057 library code and documentation. The user specifies the function for which a harness is to be
1058 generated. AutoGen gathers information for this function—such as the function body, used
1059 header files, function calling examples—from the source code and documentation. Through
1060 specific prompt templates containing the above information, an LLM is tasked with generating
1061 a new fuzz driver, while another is tasked with generating a compilation command for said
1062 driver. If the compilation fails, both LLMs are called again to fix the problem, whether it was on
1063 the driver’s or command’s side. This loop iterates until a predefined maximum value or until a
1064 fuzz driver is successfully generated and compiled. If the latter is the case, it is then executed.
1065 If execution errors exist, the LLM responsible for the driver generation is used to correct them.
1066 If not, the pipeline has terminated and a new fuzz driver has been successfully generated.

1067 5.12. Differences

1068 OverHAuL differs, in some way, with each of the aforementioned works in Chapter 5. Firstly,
1069 although KLEE and IRIS [104], [109] tackle the problem of automated testing and both IRIS and
1070 OverHAuL can be considered neurosymbolic AI tools, the similarities end there. None of them
1071 utilize LLMs the same way we do—with KLEE not utilizing them by default, as it precedes them
1072 chronologically—and neither are automating any part of the fuzzing process.

1073 When it comes to FUDGE, FuzzGen and UTopia [10]–[12], all three depend on and demand
1074 existing client code and/or unit tests. On the other hand, OverHAuL requires only the bare
1075 minimum: the library code itself. Another point of difference is that in contrast with OverHAuL,
1076 these tools operate in a linear fashion. No feedback is produced or used in any step and any
1077 point failure results in the termination of the entire run.

1078 OverHAuL challenges a common principle of these tools, stated explicitly in FUDGE’s paper
1079 [12]: “Choosing a suitable fuzz target (still) requires a human”. OverHAuL chooses to let the
1080 LLM, instead of the user, explore the available functions and choose one to target in its fuzz
1081 driver.

1082 OSS-Fuzz-Gen [9] can be considered a close counterpart of OverHAuL, and in some ways it
1083 is. A lot of inspiration was gathered from it, like for example the inclusion of static analysis
1084 and its usage in informing the LLM. Yet, OSS-Fuzz-Gen has a number of disadvantages that
1085 make it in some cases an inferior option. For one, OFG is tightly coupled with the OSS-Fuzz
1086 platform [67], which even on its own creates a plethora of issues for the common developer.
1087 To integrate their project into OSS-Fuzz, they would need to: Transform it into a ClusterFuzz
1088 project [115] and take time to write harnesses for it. Even if these prerequisites are carried
1089 out, it probably would not be enough. Per OSS-Fuzz’s documentation [114]: “To be accepted to
1090 OSS-Fuzz, an open-source project must have a significant user base and/or be critical to the
1091 global IT infrastructure”. This means that OSS-Fuzz is a viable option only for a small minority of
1092 open-source developers and maintainers. One countermeasure of the above shortcoming would
1093 be for a developer to run OSS-Fuzz-Gen locally. This unfortunately proves to be an arduous task.
1094 As it is not meant to be used standalone, OFG is not packaged in the form of a self-contained
1095 application. This makes it hard to setup and difficult to use interactively. Like in the case of
1096 FUDGE, OFG’s actions are performed linearly. No feedback is utilized nor is there graceful
1097 error handling in the case of a step’s failure. Even in the case of the experimental feature for
1098 bootstrapping unfuzzed projects, OFG’s performance varies heavily. During experimentation,
1099 a lot of generated harnesses were still wrapped either in Markdown backticks or `<code>` tags,
1100 or were accompanied with explanations inside the generated .c source file. Even if code
1101 was formatted correctly, in many cases it missed necessary headers for compilation or used
1102 undeclared functions.

1103 Lastly, the closest counterpart to OverHAuL is AutoGen [66]. Their similarity stands in the
1104 implementation of a feedback loop between LLM and generated harness. However, most other
1105 implementation decisions remain distinct. One difference regards the fuzzed function. While
1106 AutoGen requires a target function to be specified by the user in which it narrows during its
1107 whole run, OverHAuL delegates this to the LLM, letting it explore the codebase and decide by
1108 itself the best candidate. Another difference lies in the need—and the lack of—of documentation.
1109 While AutoGen requires it to gather information for the given function, OverHAuL leans into the
1110 role of a developer by reading the related code and comments and thus avoiding any mismatches
1111 between documentation and code. Finally, the LLMs’ input is built based on predefined prompt
1112 templates, a technique also present in OSS-Fuzz-Gen. OverHAuL operates one abstraction level
1113 higher, leveraging DSPy [80] for programming instead of prompting the LLMs used.

1114 In conclusion, OverHAuL constitutes an *open-source* tool that offers new functionality by
1115 offering a straightforward installation process, packaged as a self-contained Python package
1116 with minimal external dependencies. It also introduces novel approaches compared to previous
1117 work by

- 1118 1. Implementing a feedback mechanism between harness generation, compilation, and

- 1119 evaluation phases,
- 1120 2. Using autonomous ReAct agents capable of codebase exploration,
- 1121 3. Leveraging a vector store for code consumption and retrieval.

6. Future Work

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

6.1. Enhancements to Core Features

Enhancing OverHAuL’s core functionality represents a primary direction for future development. First, expanding support to encompass a wider array of build systems commonly employed in C and C++ projects—such as GNU Make, CMake, Meson, and Ninja [75], [76], [121], [122]—would significantly broaden the scope of libraries amenable to automated fuzzing using OverHAuL. This advancement would enable OverHAuL to scale effectively and be applied to larger, more complex codebases, thereby increasing its practical utility and impact.

Second, integrating additional fuzzing engines beyond LibFuzzer stands out as a strategic enhancement. Incorporation of widely adopted fuzzers like AFL++ [30] could diversify the fuzzing strategies available to OverHAuL, while exploring more experimental tools such as GraphFuzz [106] 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 [123]. Such versatility would position OverHAuL as a more universal fuzzing automation platform.

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

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

1154 enhancement would enable the tool to exploit programmer-written usage scenarios as seeds or
1155 contexts, potentially generating more meaningful and targeted fuzz inputs. Incorporating these
1156 richer information sources would likely improve the efficacy of fuzzing campaigns and uncover
1157 subtler bugs.

1158 **6.2. Experimentation with Large Language Models and Data** 1159 **Representation**

1160 OverHAuL’s reliance on large language models (LLMs) invites comprehensive experimentation
1161 with different providers and architectures to assess their comparative strengths and limitations.
1162 Conducting empirical evaluations across leading models—such as OpenAI’s o1 and o3 families
1163 and Anthropic’s Claude Opus 4—will provide valuable insights into their capabilities, cost-
1164 efficiency, and suitability for fuzz driver synthesis. Additionally, specialized code-focused LLMs,
1165 including generative and fill-in models like Codex-1 and CodeGen [54]–[56], merit exploration
1166 due to their targeted optimization for source code generation and understanding.

1167 Another dimension worthy of investigation concerns the granularity of code chunking employed
1168 during the given project’s code processing stage. Whereas the current approach partitions
1169 code at the function level, experimenting with more nuanced segmentation strategies—such as
1170 splitting per step inside a function, as a finer-grained technique—could improve the semantic
1171 coherence of stored representations and enhance retrieval relevance during fuzz driver genera-
1172 tion. This line of inquiry has the potential to optimize model input preparation and ultimately
1173 improve output quality.

1174 **6.3. Comprehensive Evaluation and Benchmarking**

1175 To thoroughly establish OverHAuL’s effectiveness, extensive large-scale evaluation beyond the
1176 initial 10-project corpus is imperative. Applying the tool to repositories indexed in the clib
1177 package manager [93], which encompasses hundreds of C libraries, would test scalability and
1178 robustness across diverse real-world settings. Such a broad benchmark would also enable sys-
1179 tematic comparisons against state-of-the-art automated fuzzing frameworks like OSS-Fuzz-Gen
1180 and AutoGen, elucidating OverHAuL’s relative strengths and identifying areas for improvement
1181 [9], [66].

1182 Complementing broad benchmarking, detailed ablation and matrix studies dissecting the con-
1183 tributions of individual pipeline components and algorithmic choices will yield critical insights
1184 into what drives OverHAuL’s performance. Understanding the impact of each module will
1185 guide targeted optimizations and support evidence-based design decisions.

1186 Furthermore, an economic analysis exploring resource consumption—such as API token usage
1187 and associated monetary costs—relative to fuzzing effectiveness would be valuable for assess-

1188 ing the practical viability of integrating LLM-based fuzz driver generation into continuous
1189 integration processes.

1190 **6.4. Practical Deployment and Community Engagement**

1191 From a usability perspective, embedding OverHAuL within a GitHub Actions workflow repre-
1192 sents a practical and impactful enhancement, enabling seamless integration with developers'
1193 existing toolchains and continuous integration pipelines. This would promote wider adoption by
1194 reducing barriers to entry and fostering real-time feedback during code development cycles.

1195 Additionally, establishing a mechanism to generate and submit automated pull requests (PRs) to
1196 the maintainers of fuzzed libraries—highlighting detected bugs and proposing patches—would
1197 not only validate OverHAuL’s findings but also contribute tangible improvements to open-
1198 source software quality. This collaborative feedback loop epitomizes the symbiosis between
1199 automated testing tools and the open-source community. As an initial step, developing targeted
1200 PRs for the projects where bugs were discovered during OverHAuL’s development would help
1201 facilitate practical follow-up and improvements.

1202 7. Conclusion

1203 Recap Performed a literature review of similar projects. Presented the algorithm *and* the
1204 implementation.

Bibliography

- 1206 [1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice-Hall Software Series).
1207 Englewood Cliffs, N.J.: Prentice-Hall, 1978, 228 pp., ISBN: 978-0-13-110163-0.
- 1208 [2] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, “The C programming language,”
1209 *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1991–2019, 1978. [Online]. Available: https://www.academia.edu/download/67840358/1978.07_Bell_System_Technical_Journal.pdf#page=85.
1210
- 1211 [3] G. J. Holzmann, “The Power of 10: Rules for Developing Safety-Critical Code,” Jun. 2006. [Online].
1212 Available: <https://web.eecs.umich.edu/~imarkov/10rules.pdf>.
- 1213 [4] Ada Developers. “Ada Reference Manual, 2022 Edition,” Ada Information Clearinghouse. (2022),
1214 [Online]. Available: https://www.adaic.org/resources/add_content/standards/22rm/html/RM-TTL.html.
1215
- 1216 [5] Rust Project Developers. “Rust Programming Language.” (2025), [Online]. Available: <https://www.rust-lang.org/>.
1217
- 1218 [6] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. “Do Users Write More Insecure Code with AI
1219 Assistants?” arXiv: 2211.03622. (Dec. 18, 2023), [Online]. Available: <http://arxiv.org/abs/2211.03622>, pre-published.
1220
- 1221 [7] N. Kosmyrna, E. Hauptmann, Y. T. Yuan, *et al.* “Your Brain on ChatGPT: Accumulation of Cognitive
1222 Debt when Using an AI Assistant for Essay Writing Task.” arXiv: 2506.08872 [cs]. (Jun. 10, 2025),
1223 [Online]. Available: <http://arxiv.org/abs/2506.08872>, pre-published.
- 1224 [8] H.-P. H. Lee, A. Sarkar, L. Tankelevitch, *et al.*, “The Impact of Generative AI on Critical Thinking:
1225 Self-Reported Reductions in Cognitive Effort and Confidence Effects From a Survey of Knowledge
1226 Workers,” 2025. [Online]. Available: https://hankhplee.com/papers/genai_critical_thinking.pdf.
- 1227 [9] D. Liu, O. Chang, J. metzman, M. Sablotny, and M. Maruseac, *OSS-fuzz-gen: Automated fuzz
1228 target generation*, version <https://github.com/google/oss-fuzz-gen/tree/v1.0>, May 2024. [Online].
1229 Available: <https://github.com/google/oss-fuzz-gen>.
- 1230 [10] B. Jeong, J. Jang, H. Yi, *et al.*, “UTopia: Automatic Generation of Fuzz Driver using Unit Tests,” in
1231 *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2676–2692. DOI: 10.1109/SP46215.
1232 2023.10179394. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10179394>.
- 1233 [11] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, “FuzzGen: Automatic fuzzer generation,” in
1234 *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287. [Online]. Available:
1235 <https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou>.
- 1236 [12] D. Babić, S. Bucur, Y. Chen, *et al.*, “FUDGE: Fuzz driver generation at scale,” in *Proceedings of the
1237 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on
1238 the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 12, 2019, pp. 975–985, ISBN:
1239 978-1-4503-5572-8. DOI: 10.1145/3338906.3340456. [Online]. Available: <https://dl.acm.org/doi/10.1145/3338906.3340456>.
1240

- [13] V. J. M. Manes, H. Han, C. Han, *et al.* “The Art, Science, and Engineering of Fuzzing: A Survey.” arXiv: [1812.00140 \[cs\]](https://arxiv.org/abs/1812.00140). (Apr. 7, 2019), [Online]. Available: <http://arxiv.org/abs/1812.00140>, pre-published.
- [14] A. Takanen, J. DeMott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance* (Information Security and Privacy Library), Second edition. Boston London Norwood, MA: Artech House, 2018, 1 p., ISBN: 978-1-63081-519-6.
- [15] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Upper Saddle River, NJ: Addison-Wesley, 2007, 543 pp., ISBN: 978-0-321-44611-4.
- [16] N. Rathaus and G. Evron, *Open Source Fuzzing Tools*, G. Evron, Ed. Burlington, MA: Syngress Pub, 2007, 199 pp., ISBN: 978-1-59749-195-2.
- [17] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1, 1990, ISSN: 0001-0782. DOI: [10.1145/96267.96279](https://doi.org/10.1145/96267.96279). [Online]. Available: <https://dl.acm.org/doi/10.1145/96267.96279>.
- [18] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318. [Online]. Available: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>.
- [19] LLVM Project. “libFuzzer – a library for coverage-guided fuzz testing. — LLVM 21.0.0git documentation.” (2025), [Online]. Available: <https://llvm.org/docs/LibFuzzer.html>.
- [20] A. Rebert, S. K. Cha, T. Avgerinos, *et al.*, “Optimizing seed selection for fuzzing,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14, USA: USENIX Association, Aug. 20, 2014, pp. 861–875, ISBN: 978-1-931971-15-7.
- [21] OWASP Foundation. “Fuzzing.” (), [Online]. Available: <https://owasp.org/www-community/Fuzzing>.
- [22] Blackduck, Inc. “Heartbleed Bug.” (Mar. 7, 2025), [Online]. Available: <https://heartbleed.com/>.
- [23] CVE Program. “CVE - CVE-2014-0160.” (2014), [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160>.
- [24] The OpenSSL Project, *Openssl/openssl*, OpenSSL, Jul. 15, 2025. [Online]. Available: <https://github.com/openssl/openssl>.
- [25] D. Wheeler. “How to Prevent the next Heartbleed.” (2014), [Online]. Available: <https://dwheeler.com/essays/heartbleed.html>.
- [26] GNU Project. “Bash - GNU Project - Free Software Foundation.” (), [Online]. Available: <https://www.gnu.org/software/bash/>.
- [27] M. Zalewski. “American fuzzy lop.” (), [Online]. Available: <https://lcamtuf.coredump.cx/afl/>.
- [28] J. Saarinen. “Further flaws render Shellshock patch ineffective,” *iTnews*. (Sep. 29, 2014), [Online]. Available: <https://www.itnews.com.au/news/further-flaws-render-shellshock-patch-ineffective-396256>.
- [29] T. Simonite, “This Bot Hunts Software Bugs for the Pentagon,” *Wired*, Jun. 1, 2020, ISSN: 1059-1028. [Online]. Available: <https://www.wired.com/story/bot-hunts-software-bugs-pentagon/>.
- [30] M. Heuse, H. Eißfeldt, A. Fioraldi, and D. Maier, *AFL++*, version 4.00c, Jan. 2022. [Online]. Available: <https://github.com/AFLplusplus/AFLplusplus>.
- [31] LLVM Project. “The LLVM Compiler Infrastructure Project.” (2025), [Online]. Available: <https://llvm.org/>.

- [32] F. Bellard, P. Maydell, and QEMU Team, *QEMU*, version 10.0.2, May 29, 2025. [Online]. Available: <https://www.qemu.org/>.
- [33] Unicorn Engine, *Unicorn-engine/unicorn*, Unicorn Engine, Jul. 15, 2025. [Online]. Available: <https://github.com/unicorn-engine/unicorn>.
- [34] H. Li, “Language models: Past, present, and future,” *Commun. ACM*, vol. 65, no. 7, pp. 56–63, Jun. 21, 2022, issn: 0001-0782. doi: [10.1145/3490443](https://doi.org/10.1145/3490443). [Online]. Available: <https://dl.acm.org/doi/10.1145/3490443>.
- [35] Z. Wang, Z. Chu, T. V. Doan, S. Ni, M. Yang, and W. Zhang, “History, development, and principles of large language models: An introductory survey,” *AI and Ethics*, vol. 5, no. 3, pp. 1955–1971, Jun. 1, 2025, issn: 2730-5961. doi: [10.1007/s43681-024-00583-7](https://doi.org/10.1007/s43681-024-00583-7). [Online]. Available: <https://doi.org/10.1007/s43681-024-00583-7>.
- [36] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate.” arXiv: [1409.0473](https://arxiv.org/abs/1409.0473) [cs, stat]. (May 19, 2016), [Online]. Available: <http://arxiv.org/abs/1409.0473>, pre-published.
- [37] A. Vaswani, N. Shazeer, N. Parmar, *et al.* “Attention Is All You Need.” arXiv: [1706.03762](https://arxiv.org/abs/1706.03762) [cs]. (Aug. 1, 2023), [Online]. Available: <http://arxiv.org/abs/1706.03762>, pre-published.
- [38] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” arXiv: [1810.04805](https://arxiv.org/abs/1810.04805) [cs]. (May 24, 2019), [Online]. Available: <http://arxiv.org/abs/1810.04805>, pre-published.
- [39] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” 2018. [Online]. Available: <https://www.mikecaptain.com/resources/pdf/GPT-1.pdf>.
- [40] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019. [Online]. Available: <https://storage.prod.researchhub.com/uploads/papers/2020/06/01/language-models.pdf>.
- [41] T. B. Brown, B. Mann, N. Ryder, *et al.* “Language Models are Few-Shot Learners.” arXiv: [2005.14165](https://arxiv.org/abs/2005.14165) [cs]. (Jul. 22, 2020), [Online]. Available: <http://arxiv.org/abs/2005.14165>, pre-published.
- [42] OpenAI, J. Achiam, S. Adler, *et al.* “GPT-4 Technical Report.” arXiv: [2303.08774](https://arxiv.org/abs/2303.08774) [cs]. (Mar. 4, 2024), [Online]. Available: <http://arxiv.org/abs/2303.08774>, pre-published.
- [43] Anthropic. “Claude.” (2025), [Online]. Available: <https://claude.ai/new>.
- [44] DeepSeek-AI, D. Guo, D. Yang, *et al.* “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.” arXiv: [2501.12948](https://arxiv.org/abs/2501.12948) [cs]. (Jan. 22, 2025), [Online]. Available: <http://arxiv.org/abs/2501.12948>, pre-published.
- [45] A. Grattafiori, A. Dubey, A. Jauhri, *et al.* “The Llama 3 Herd of Models.” arXiv: [2407.21783](https://arxiv.org/abs/2407.21783) [cs]. (Nov. 23, 2024), [Online]. Available: <http://arxiv.org/abs/2407.21783>, pre-published.
- [46] OpenAI. “ChatGPT.” (2025), [Online]. Available: <https://chatgpt.com>.
- [47] Google. “Google Gemini,” Gemini. (2025), [Online]. Available: <https://gemini.google.com>.
- [48] J. Wei, X. Wang, D. Schuurmans, *et al.* “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.” arXiv: [2201.11903](https://arxiv.org/abs/2201.11903) [cs]. (Jan. 10, 2023), [Online]. Available: <http://arxiv.org/abs/2201.11903>, pre-published.
- [49] S. Yao, D. Yu, J. Zhao, *et al.* “Tree of Thoughts: Deliberate Problem Solving with Large Language Models.” arXiv: [2305.10601](https://arxiv.org/abs/2305.10601) [cs]. (Dec. 3, 2023), [Online]. Available: <http://arxiv.org/abs/2305.10601>, pre-published.

- [50] P. Lewis, E. Perez, A. Piktus, *et al.* "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." arXiv: 2005.11401 [cs]. (Apr. 12, 2021), [Online]. Available: <http://arxiv.org/abs/2005.11401>, pre-published.
- [51] S. Yao, J. Zhao, D. Yu, *et al.* "ReAct: Synergizing Reasoning and Acting in Language Models." arXiv: 2210.03629. (Mar. 10, 2023), [Online]. Available: <http://arxiv.org/abs/2210.03629>, pre-published.
- [52] Anysphere. "Cursor - The AI Code Editor." (2025), [Online]. Available: <https://cursor.com/>.
- [53] Microsoft. "GitHub Copilot · Your AI pair programmer," GitHub. (2025), [Online]. Available: <https://github.com/features/copilot>.
- [54] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "CodeGen2: Lessons for training llms on programming and natural languages," *ICLR*, 2023.
- [55] E. Nijkamp, B. Pang, H. Hayashi, *et al.*, "CodeGen: An open large language model for code with multi-turn program synthesis," *ICLR*, 2023.
- [56] OpenAI. "Introducing Codex." (May 16, 2025), [Online]. Available: <https://openai.com/index/introducing-codex/>.
- [57] A. Sarkar and I. Drosos. "Vibe coding: Programming through conversation with artificial intelligence." arXiv: 2506.23253 [cs]. (Jun. 29, 2025), [Online]. Available: <http://arxiv.org/abs/2506.23253>, pre-published.
- [58] A. Sheth, K. Roy, and M. Gaur. "Neurosymbolic AI – Why, What, and How." arXiv: 2305.00813 [cs]. (May 1, 2023), [Online]. Available: <http://arxiv.org/abs/2305.00813>, pre-published.
- [59] A. d'Avila Garcez and L. C. Lamb. "Neurosymbolic AI: The 3rd Wave." arXiv: 2012.05876. (Dec. 16, 2020), [Online]. Available: <http://arxiv.org/abs/2012.05876>, pre-published.
- [60] D. Ganguly, S. Iyengar, V. Chaudhary, and S. Kalyanaraman. "Proof of Thought : Neurosymbolic Program Synthesis allows Robust and Interpretable Reasoning." arXiv: 2409.17270. (Sep. 25, 2024), [Online]. Available: <http://arxiv.org/abs/2409.17270>, pre-published.
- [61] M. Gaur and A. Sheth. "Building Trustworthy NeuroSymbolic AI Systems: Consistency, Reliability, Explainability, and Safety." arXiv: 2312.06798. (Dec. 5, 2023), [Online]. Available: <http://arxiv.org/abs/2312.06798>, pre-published.
- [62] D. Tilwani, R. Venkataramanan, and A. P. Sheth. "Neurosymbolic AI approach to Attribution in Large Language Models." arXiv: 2410.03726. (Sep. 30, 2024), [Online]. Available: <http://arxiv.org/abs/2410.03726>, pre-published.
- [63] M. K. Sarker, L. Zhou, A. Eberhart, and P. Hitzler, "Neuro-symbolic artificial intelligence: Current trends," *AI Communications*, vol. 34, no. 3, pp. 197–209, Mar. 4, 2022, issn: 1875-8452, 0921-7126. doi: 10.3233/aic-210084. [Online]. Available: <https://journals.sagepub.com/doi/full/10.3233/AIC-210084>.
- [64] H. Kautz, "The Third AI Summer," Lecture, presented at the 34th Annual Meeting of the Association for the Advancement of Artificial Intelligence (New York, NY, USA), Feb. 10, 2020. [Online]. Available: https://www.youtube.com/watch?v=_cQITY0SPiw.
- [65] L. Torvalds, *Git*, Apr. 7, 2005. [Online]. Available: <https://git-scm.com/>.
- [66] Y. Sun, "Automated Generation and Compilation of Fuzz Driver Based on Large Language Models," in *Proceedings of the 2024 9th International Conference on Cyber Security and Information Engineering*, ser. ICCSIE '24, New York, NY, USA: Association for Computing Machinery, Dec. 3, 2024, pp. 461–468, isbn: 979-8-4007-1813-7. doi: 10.1145/3689236.3689272. [Online]. Available: <https://doi.org/10.1145/3689236.3689272>.

- [67] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu, *OSS-Fuzz*, Apr. 8, 2025. [Online]. Available: <https://github.com/google/oss-fuzz>.
- [68] T. Mikolov, K. Chen, G. Corrado, and J. Dean. “Efficient Estimation of Word Representations in Vector Space.” arXiv: [1301.3781](https://arxiv.org/abs/1301.3781) [cs]. (Sep. 6, 2013), [Online]. Available: [http://arxiv.org/abs/1301.3781](https://arxiv.org/abs/1301.3781), pre-published.
- [69] Open Source Security Foundation (OpenSSF), *OSSF/fuzz-introspector*, Open Source Security Foundation (OpenSSF), Jun. 30, 2025. [Online]. Available: <https://github.com/ossf/fuzz-introspector>.
- [70] Python Software Foundation. “Venv — Creation of virtual environments,” Python documentation. (Jul. 17, 2025), [Online]. Available: <https://docs.python.org/3/library/venv.html>.
- [71] pip developers. “Pip documentation v25.1.1.” (2025), [Online]. Available: <https://pip.pypa.io/en/stable/>.
- [72] D. A. Wheeler. “Flawfinder Home Page,” Flawfinder. (), [Online]. Available: <https://dwheeler.com/flawfinder/>.
- [73] S. Zhao, Y. Yang, Z. Wang, Z. He, L. K. Qiu, and L. Qiu. “Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely.” arXiv: [2409.14924](https://arxiv.org/abs/2409.14924) [cs]. (Sep. 23, 2024), [Online]. Available: [http://arxiv.org/abs/2409.14924](https://arxiv.org/abs/2409.14924), pre-published.
- [74] M. Chen, J. Tworek, H. Jun, *et al.* “Evaluating Large Language Models Trained on Code.” arXiv: [2107.03374](https://arxiv.org/abs/2107.03374) [cs]. (Jul. 14, 2021), [Online]. Available: [http://arxiv.org/abs/2107.03374](https://arxiv.org/abs/2107.03374), pre-published.
- [75] A. Cedilnik, B. Hoffman, B. King, K. Martin, and A. Neundorff, *CMake - Upgrade Your Software Build System*, 2000. [Online]. Available: <https://cmake.org/>.
- [76] S. I. Feldman, “Make — a program for maintaining computer programs,” *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979, issn: 1097-024X. doi: [10.1002/spe.4380090402](https://doi.org/10.1002/spe.4380090402). [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402>.
- [77] T. He, *Sighingnow/libclang*, Jul. 3, 2025. [Online]. Available: <https://github.com/sighingnow/libclang>.
- [78] OpenAI Docs. “Text-embedding-3-small - OpenAI API.” (2025), [Online]. Available: <https://platform.openai.com>.
- [79] M. Douze, A. Guzhva, C. Deng, *et al.* “The Faiss library.” arXiv: [2401.08281](https://arxiv.org/abs/2401.08281) [cs]. (Feb. 11, 2025), [Online]. Available: [http://arxiv.org/abs/2401.08281](https://arxiv.org/abs/2401.08281), pre-published.
- [80] O. Khattab, A. Singhvi, P. Maheshwari, *et al.* “DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines.” arXiv: [2310.03714](https://arxiv.org/abs/2310.03714) [cs]. (Oct. 5, 2023), [Online]. Available: [http://arxiv.org/abs/2310.03714](https://arxiv.org/abs/2310.03714), pre-published.
- [81] Stanford NLP Team. “Signatures - DSPy Documentation.” (2025), [Online]. Available: <https://dspy.ai/learn/programming/signatures/>.
- [82] Stanford NLP Team. “ReAct - DSPy Documentation.” (2025), [Online]. Available: <https://dspy.ai/api/modules/ReAct/>.
- [83] H. Chase, *LangChain*, Oct. 2022. [Online]. Available: <https://github.com/langchain-ai/langchain>.
- [84] J. Liu, *LlamaIndex*, Nov. 2022. doi: [10.5281/zenodo.1234](https://doi.org/10.5281/zenodo.1234). [Online]. Available: https://github.com/jerryliu/llama_index.
- [85] F. Both. “Why we no longer use LangChain for building our AI agents.” (2024), [Online]. Available: <https://octomind.dev/blog/why-we-no-longer-use-langchain-for-building-our-ai-agents>.

- [86] M. Woolf. “The Problem With LangChain.” (Jul. 14, 2023), [Online]. Available: <https://minimaxir.com/2023/07/langchain-problem/>.
- [87] Woyera. “6 Reasons why Langchain Sucks,” Medium. (Sep. 8, 2023), [Online]. Available: <https://medium.com/@woyera/6-reasons-why-langchain-sucks-b6c99c98efbe>.
- [88] Astral, *Astral-sh/ruff*, Astral, Jul. 18, 2025. [Online]. Available: <https://github.com/astral-sh/ruff>.
- [89] Astral, *Astral-sh/uv*, Astral, Jul. 18, 2025. [Online]. Available: <https://github.com/astral-sh/uv>.
- [90] A. Cortesi, M. Hils, and T. Kriechbaumer, *Mitmproxy/pdoc*, mitmproxy, Jul. 18, 2025. [Online]. Available: <https://github.com/mitmproxy/pdoc>.
- [91] PyTest Dev Team, *Pytest-dev/pytest*, pytest-dev, Jul. 18, 2025. [Online]. Available: <https://github.com/pytest-dev/pytest>.
- [92] Python Software Foundation, *Python/mypy*, Python, Jul. 18, 2025. [Online]. Available: <https://github.com/python/mypy>.
- [93] Clibs Project. “Clib Packages,” GitHub. (2025), [Online]. Available: <https://github.com/clibs/clib/wiki/Packages>.
- [94] Clibs Project, *Clibs/clib*, clibs, Jul. 1, 2025. [Online]. Available: <https://github.com/clibs/clib>.
- [95] OpenAI Docs. “GPT-4.1 mini - Open AI API.” (2025), [Online]. Available: <https://platform.openai.com>.
- [96] GitHub Docs. “About GitHub-hosted runners,” GitHub Docs. (2025), [Online]. Available: <https://docs-internal.github.com/en/actions/concepts/runners/about-github-hosted-runners>.
- [97] GitHub Docs. “Choosing the runner for a job,” GitHub Docs. (2025), [Online]. Available: <https://docs-internal.github.com/en/actions/how-tos/writing-workflows/choosing-where-your-workflow-runs/choosing-the-runner-for-a-job>.
- [98] O. I. Franksen, “Babbage and cryptography. Or, the mystery of Admiral Beaufort’s cipher,” *Mathematics and Computers in Simulation*, vol. 35, no. 4, pp. 327–367, 1993. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/037847549390063Z>.
- [99] F. Bacon, *Of the Proficiency and Advancement of Learning... Edited by the Rev. GW Kitchin*. Bell & Daldy, 1861.
- [100] T. Preston-Werner. “Semantic Versioning 2.0.0,” Semantic Versioning. (), [Online]. Available: <https://semver.org/>.
- [101] D. Giannone. “Demystifying AI Agents: ReAct-Style Agents vs Agentic Workflows,” Medium. (Feb. 9, 2025), [Online]. Available: <https://medium.com/@DanGiannone/demystifying-ai-agents-react-style-agents-vs-agentic-workflows-cedca7e26471>.
- [102] OpenAI Docs. “Model optimization - OpenAI API.” (2025), [Online]. Available: <https://platform.openai.com>.
- [103] S. Kim and S.-y. Lee, “Performance Comparison of Prompt Engineering and Fine-Tuning Approaches for Fuzz Driver Generation Using Large Language Models,” in *Innovative Mobile and Internet Services in Ubiquitous Computing*, L. Barolli, H.-C. Chen, and K. Yim, Eds., Cham: Springer Nature Switzerland, 2025, pp. 111–120, ISBN: 978-3-031-96093-2. DOI: [10.1007/978-3-031-96093-2_12](https://doi.org/10.1007/978-3-031-96093-2_12).
- [104] Z. Li, S. Dutta, and M. Naik. “IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities.” arXiv: [2405.17238](https://arxiv.org/abs/2405.17238) [cs]. (Apr. 6, 2025), [Online]. Available: <http://arxiv.org/abs/2405.17238>, pre-published.

- [105] D. Wang, G. Zhou, L. Chen, D. Li, and Y. Miao. “ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model.” arXiv: [2409.00922](https://arxiv.org/abs/2409.00922) [cs]. (Sep. 1, 2024), [Online]. Available: <http://arxiv.org/abs/2409.00922>, pre-published.
- [106] H. Green and T. Avgerinos, “GraphFuzz: Library API fuzzing with lifetime-aware dataflow graphs,” in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 1070–1081. doi: [10.1145/3510003.3510228](https://doi.org/10.1145/3510003.3510228). [Online]. Available: <https://dl.acm.org/doi/10.1145/3510003.3510228>.
- [107] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang. “Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT.” arXiv: [2304.02014](https://arxiv.org/abs/2304.02014) [cs]. (Apr. 4, 2023), [Online]. Available: <http://arxiv.org/abs/2304.02014>, pre-published.
- [108] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, New York, NY, USA: Association for Computing Machinery, Jul. 13, 2023, pp. 423–435, ISBN: 979-8-4007-0221-1. doi: [10.1145/3597926.3598067](https://doi.org/10.1145/3597926.3598067). [Online]. Available: <https://dl.acm.org/doi/10.1145/3597926.3598067>.
- [109] 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>.
- [110] N. Sasirekha, A. Edwin Robert, and M. Hemalatha, “Program Slicing Techniques and its Applications,” *International Journal of Software Engineering & Applications*, vol. 2, no. 3, pp. 50–64, Jul. 31, 2011, ISSN: 09762221. doi: [10.5121/ijsea.2011.2304](https://doi.org/10.5121/ijsea.2011.2304). [Online]. Available: <http://www.airccse.org/journal/ijsea/papers/0711ijsea04.pdf>.
- [111] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang. “IntelliGen: Automatic Driver Synthesis for FuzzTesting.” arXiv: [2103.00862](https://arxiv.org/abs/2103.00862) [cs]. (Mar. 1, 2021), [Online]. Available: <http://arxiv.org/abs/2103.00862>, pre-published.
- [112] H. Xu, W. Ma, T. Zhou, *et al.* “CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph.” arXiv: [2411.11532](https://arxiv.org/abs/2411.11532) [cs]. (Dec. 20, 2024), [Online]. Available: <http://arxiv.org/abs/2411.11532>, pre-published.
- [113] Y. Lyu, Y. Xie, P. Chen, and H. Chen. “Prompt Fuzzing for Fuzz Driver Generation.” arXiv: [2312.17677](https://arxiv.org/abs/2312.17677) [cs]. (May 29, 2024), [Online]. Available: <http://arxiv.org/abs/2312.17677>, pre-published.
- [114] OSS-Fuzz. “OSS-Fuzz Documentation,” OSS-Fuzz. (2025), [Online]. Available: <https://github.io/oss-fuzz/>.
- [115] Google, *Google/clusterfuzz*, Google, Apr. 9, 2025. [Online]. Available: <https://github.com/google/clusterfuzz>.
- [116] Google, *Google/fuzztest*, Google, Jul. 10, 2025. [Online]. Available: <https://github.com/google/fuzztest>.
- [117] Google, *Google/honggfuzz*, Google, Jul. 10, 2025. [Online]. Available: <https://github.com/google/honggfuzz>.

- 1498 [118] D. Liu, J. Metzman, O. Chang, and G. O. S. S. Team. “AI-Powered Fuzzing: Breaking the Bug
1499 Hunting Barrier,” Google Online Security Blog. (Aug. 16, 2023), [Online]. Available: [https://
1500 security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html](https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html).
- 1501 [119] L. Thomason, *Leethomason/tinysql2*, Jul. 10, 2025. [Online]. Available: [https://github.com/
1502 leethomason/tinysql2](https://github.com/leethomason/tinysql2).
- 1503 [120] OSS-Fuzz Maintainers. “Introducing LLM-based harness synthesis for unfuzzed projects,” OSS-
1504 Fuzz blog. (May 27, 2024), [Online]. Available: [https://blog.oss-fuzz.com/posts/introducing-llm-
1505 based-harness-synthesis-for-unfuzzed-projects/](https://blog.oss-fuzz.com/posts/introducing-llm-based-harness-synthesis-for-unfuzzed-projects/).
- 1506 [121] E. Martin, *Ninja-build/ninja*, ninja-build, Jul. 14, 2025. [Online]. Available: [https://github.com/
1507 ninja-build/ninja](https://github.com/ninja-build/ninja).
- 1508 [122] J. Pakkanen, *Mesonbuild/meson*, The Meson Build System, Jul. 14, 2025. [Online]. Available:
1509 <https://github.com/mesonbuild/meson>.
- 1510 [123] Google, *Google/atheris*, Google, Apr. 9, 2025. [Online]. Available: [https://github.com/google/
1511 atheris](https://github.com/google/atheris).

1512 A. Abandoned Techniques

1513 During its development, OverHAuL went through several iterations. A number of approaches
1514 were implemented and evaluated, with some being replaced for better alternatives. These are:

1515 1. One-shot harness generation

1516 Before the iterative feedback loop (Section 3.3.1) was implemented, OverHAuL attempted
1517 to operate in a straightforward pipeline, with just a “generator” agent being tasked to
1518 generate the harness. This meant that at either the compilation step or evaluation step,
1519 any failure resulted in the execution being terminated. This approach put too much
1520 responsibility in the response of a single LLM query, with results more often than not
1521 being unsatisfactory.

1522 2. Chain-of-Thought LLM instances

1523 The current implementation of ReAct agents has effectively supplanted the less effective
1524 Chain of Thought (COT) LLM modules [48]. This shift underscores a critical realization
1525 in the harness generation process: the primary challenge lies not in the creation of the
1526 harness itself, but rather in the necessity for real-time feedback during execution. This is
1527 the reason why first employing COT prompting offered limited observed improvements.

1528 COT techniques are particularly advantageous when the task assigned to the LLM de-
1529 mands a more reflective, in-depth analysis. However, when it comes to tasks such as
1530 knowledge extraction from a codebase oracle and taking live feedback from the environ-
1531 ment into consideration, ReAct agents demonstrate greater efficiency and effectiveness.

1532 3. Source code concatenation

1533 Initially, there was no implementation of a codebase oracle. Instead, the LLM agents
1534 operated with a Python string that contained a concatenation of all the collected source
1535 code. While this method proved effective for smaller and simpler projects, it encountered
1536 significant limitations when applied to more complex codebases. The primary challenge
1537 was the excessive consumption of the LLM’s context window, which hindered its ability
1538 to process and analyze larger codebases effectively. As a result, this approach became
1539 increasingly unsustainable as project complexity grew, underscoring the need for a more
1540 robust solution.

1541 4. {index, read}_tool usage for ReAct agents

1542 The predecessor of the oracle comprised a dual-system approach for code exploration,
1543 integrating the index_tool and the read_tool. The index_tool offered the LLM agent a

1544 structured JSON object that delineated the project’s architecture, including all relevant
1545 file paths. On the other hand, the `read_tool` required a file path as input and returned the
1546 file’s content, albeit truncated to a maximum of 4000 characters. While this methodology
1547 presented an improvement in scalability over earlier systems, several limitations persisted.

1548 Firstly, the LLM was constrained to searching through the codebase strictly in file-specific
1549 terms, which limited its efficacy in understanding the broader context of code relationships.
1550 Furthermore, the imposed character limit on the `read_tool` meant that certain portions of
1551 the codebase remained inaccessible, impeding the agent’s analytical capabilities. Even if
1552 this character limit were to be lifted, the resultant output would still occupy a significant
1553 portion of the context window, particularly in larger and more intricate projects. As such,
1554 while this approach offered advancements in code exploration, it still fell short.

1555 B. Sample Generated Harnesses

1556 This appendix presents a selection of harnesses generated by OverHAuL, originating from the
1557 benchmark run [16351915455](#). The examples included herein demonstrate a notable diversity,
1558 ranging from concise and straightforward harnesses to those that are more elaborate and
1559 complex. The crash inputs are provided in the form of their corresponding xxd hexadecimal
1560 representations.

1561 B.1. `clibs/buffer`

1562 Generated harness:

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

```

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

```

1563 Body of function that the harness targets:

```

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

```



```

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

```

1564 Crash input which triggers an exception in buffer_appendf()>vsnprintf():

```

1565 00000000: 1b2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d  .-----
1566 00000010: 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d  -----
1567 00000020: 2d2d 2d2d 2d2d 2d2d 2d2d 2d25 2d2d 2d2d  -----%-----
1568 00000030: 2d2d 2d2d 2d2d 2d2d 2d2d 2d2d 2d      -----

```

1569 B.2. willem/cbuffer

1570 Generated harness:

```

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

```

```

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

```

```

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

```

```

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

```

1571 Body of function that the harness targets:

```

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

```

1572 Crash input which triggers an exception in cbuf_offer()>memcpy():

1573 00000000: 0a .

1574 B.3. dvhar/dateparse

1575 Generated harness:

```

1 #include <stddef.h>
2 #include <stdint.h>
3 #include "dateparse.h"
4
5 // LibFuzzer entry point
6 int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
7     if (size == 0) return 0;
8
9 }

```

```

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

```

1576 Body of function that the harness targets:

```

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

```

1577 Crash input that causes a heap buffer overflow in dateparse()>parseTime()>setMonth()>strncpy().

1578 00000000: 6632 2066 f2 f

1579 B.4. h2non/semver.c

1580 Generated harness:

```

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

```

```

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

```

```
56 }
```

1581 Bodies of functions that the harness targets:

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

```

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

```

1582 Crash input that causes a stack buffer overflow in semver_render()>concat_char()>sprintf():

```

1583 00000000: 392d 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 9-+++++
1584 00000010: 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b +++++
1585 00000020: 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b +++++
1586 00000030: 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b 2b2b +++++
1587 00000040: 2b2b 2b2b 2b2b 2b46 4c                +++++FL

```


C. DSPy Custom Signatures

```

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

```

```

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

```

```

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

```