OverHAuL

Harnessing Automation for C Libraries via LLMs

Konstantinos Chousos

July, 2025

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

Preface

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

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

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

Acknowledgments

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

Table of contents

1 Introduction

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

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

1.1 Motivation

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

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

1.2 Preview of following sections (rename)

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

2 Background

quarto definitions

2.1 Fuzzing

Discovering vulnerabilities in the development stage instead of in production.

Discovering vulnerabilities ourselves before attackers do.

Borrowing definitions from [1]:

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.

Definition 2.2 (Fuzz Testing)**.** Fuzz testing is the use of fuzzing to test if a PUT violates a security policy.

Definition 2.3 (Fuzzer)**.** A fuzzer is a program that performs fuzz testing on a PUT.

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

Definition 2.5 (Bug Oracle)**.** A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific security policy.

Definition 2.6 (Black-box fuzzer)**.** A black-box fuzzer is a testing tool that inputs random or specified data into a software application without knowledge of its internal workings, aiming to uncover vulnerabilities or bugs by observing the program's behavior in response to various inputs.

Definition 2.7 (White-box fuzzer)**.** A white-box fuzzer is a testing tool that analyzes the internal structure and logic of a program to generate test inputs. It uses knowledge of the code, such as control flow and data paths, to systematically explore all possible execution paths and identify vulnerabilities more effectively.

Definition 2.8 (Grey-box fuzzer)**.** A grey-box fuzzer is a testing tool that combines aspects of both black-box and white-box fuzzing. It has limited knowledge of the internal workings of the application, often using some code coverage information or program analysis to generate more targeted inputs, thereby improving the efficiency of vulnerability detection.

Definition 2.9 (Generational fuzzing)**.** Generationbased fuzzers produce test cases based on a given model that describes the inputs expected by the PUT, e.g. a Backus–Naur form (BNF) grammar [2].

Definition 2.10 (Mutational fuzzing)**.** mutation-based fuzzers produce test cases by mutating a given seed input.

terminology: fuzz campaign (Definition 2.4), harness, driver, target, corpus

Why fuzz?

2.1.1 Fuzzing examples

Heartbleed [3], shellshock [4].

2.1.2 Fuzzer engines

C/C++: AFL [5] & AFL++ [5, pp. ++]. LibFuzzer [6].

Python: Atheris [7].

Java, Rust etc…

An example of a fuzz target/harness can be seen in Listing 2.1 [6].

OSS-Fuzz: 2016, after heartbleed.

2.2 Large Language Models (LLMs)

Transformers [8], 2017–2025. ChatGPT/OpenAI history & context. Claude, Llama (1–3) etc.

Listing 2.1 A simple function that does something interesting if it receives the input “HI!”.

```
cat << EOF > test_fuzzer.cc
#include <stdint.h>
#include <stddef.h>
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  if (size > 0 && data[0] == 'H')
    if (size > 1 && data[1] == 'I')
       if (size > 2 && data[2] == '!')
       __builtin_trap();
  return 0;
}
EOF
# Build test_fuzzer.cc with asan and link against libFuzzer.
clang++ -fsanitize=address,fuzzer test_fuzzer.cc
# Run the fuzzer with no corpus.
./a.out
```

2.2.1 Prompting

Prompting techniques.

1. Zero-shot.
2. One-shot.
3. Chain of Thought [9].
4. ReACt [10].
5. Tree of Thoughts [11].

Comparison, strengths weaknesses etc. [12].

[13]

2.2.2 LLM Programming Libraries (?)

Langchain & LangGraph, LlamaIndex [14]–[16]. DSPy [17].

Comparison, relevance to our usecase.

2.3 Neurosymbolic AI

TODO [18]–[23].

3 Related work

Automated testing, automated fuzzing and automated harness creation have a long research history. Still, a lot of ground remains to be covered until true automation of these tasks is achieved. Until the introduction of transformers [8] and the 2020's boom of commercial GPTs [24], automation regarding testing and fuzzing was mainly attempted through static and dynamic program analysis methods. These approaches are still utilized, but the fuzzing community has shifted almost entirely to researching the incorporation and employment of LLMs in the last half decade, in the name of automation [25]–[34].

3.1 Previous projects

3.1.1 KLEE

KLEE [35] 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 [36] bytecode representation of programs, allowing it to be applied to a wide range of C programs compiled to the LLVM intermediate representation.

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

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

3.1.2 IRIS

IRIS [25] is a 2025 open-source neurosymbolic system for static vulnerability analysis. Given a codebase and a list of user-specified Common Weakness Enumerations (CWEs), it analyzes source code to identify paths that may correspond to known vulnerability classes. IRIS combines symbolic analysis—such as control- and data-flow reasoning—with neural models trained to generalize over code patterns. It outputs candidate vulnerable paths along with explanations and CWE references. The system operates on full repositories and supports extensible CWE definitions.

3.1.3 FUDGE

FUDGE [34] is a closed-source tool, made by Google, for automatic harness generation of C and C++ projects based on existing client code. It was used in conjunction with and in the improvement of Google's OSS-Fuzz [37]. Being deployed inside Google's infrastructure, FUDGE continuously examines Google's internal code repository, searching for code that uses external libraries in a meaningful and "fuzzable" way (i.e. predominantly for parsing). If found, such code is **sliced** [38], per FUDGE, based on its Abstract Syntax Tree (AST) using LLVM's Clang tool [36]. The above process results in a set of abstracted mostly-self-contained code snippets that make use of a library's calls and/or API. These snippets are later **synthesized** into the body of a fuzz driver, with variables being replaced and the fuzz input being utilized. Each is then injected in an `LLVMFuzzerTestOneInput` function and finalized as a fuzzing harness. A building and evaluation phase follows for each harness, where they are executed and examined. Every passing harness along with its evaluation results is stored in FUDGE's database, reachable to the user through a custom web-based UI.

3.1.4 UTopia

UTopia [30] (stylized UTopia) is another open-source automatic harness generation framework. Aside from the library code, It operates solely on user-provided unit tests since, according to Jeong, Jang, Yi, *et al.* [30], they are a resource of complete and correct API usage examples containing working library set-ups and tear-downs. Additionally, each of them are already close to a fuzz target, in the sense that they already examine a single and self-contained API usage pattern. Each generated harness follows the same data flow of the originating unit test. Static analysis is employed to figure out what fuzz input placement would yield the most results. It is also utilized in abstracting the tests away from the syntactical differences between testing frameworks, along with slicing and AST traversing using Clang.

3.1.5 FuzzGen

Another project of Google is FuzzGen [33], this time open-source. Like FUDGE, it leverages existing client code of the target library to create fuzz targets for it. FuzzGen uses whole-system analysis, through which it creates an *Abstract API Dependence Graph* (A^2DG). It uses the latter to automatically generate LibFuzzer-compatible harnesses. For FuzzGen to work, the user needs to provide both client code and/or tests for the API and the API library's source code as well. FuzzGen uses the client code to infer the *correct usage* of the API and not its general structure, in contrast to FUDGE. FuzzGen's workflow can be divided into three phases: **1. API usage inference**. By consuming and analyzing client code and tests that concern the library under test, FuzzGen recognizes which functions belong to the library and learns its correct API usage patterns. This process is done with the help of Clang. To test if a function is actually a part of the library, a sample program is created that uses it. If the program compiles successfully, then the function is indeed a valid API call. **2. A^2DG construction mechanism**. For all the existing API calls, FuzzGen builds an A^2DG to record the API usages and infers its intended structure. After completion, this directed graph contains all the valid API call sequences found in the client code corpus. It is built in a two-step process: First, many smaller A^2DGs are created, one for each root function per client code snippet. Once such graphs have been created for all the available client code instances, they are combined to formulate the master A^2DG. This graph can be seen as a template for correct usage of the library. **3. Fuzzer generator**. Through the A^2DG, a fuzzing harness is created. Contrary to FUDGE, FuzzGen does not create multiple "simple" harnesses but a single complex one with the goal of covering the whole of the A^2DG. In other words, while FUDGE fuzzes a single API call at a time, FuzzGen's result is a single harness that tries to fuzz the given library all at once through complex API usage.

3.1.6 OSS-Fuzz

OSS-Fuzz [37], [39] is a continuous, scalable and distributed cloud fuzzing solution for critical and prominent open-source projects. Developers of such software can submit their projects to OSS-Fuzz's platform, where its harnesses are built and constantly executed. This results in multiple bug findings that are later disclosed to the primary developers and are later patched.

OSS-Fuzz started operating in 2016, an initiative in response to the Heartbleed vulnerability [3], [40], [41]. Its hope is that through more extensive fuzzing such errors could be caught and corrected before having the chance to be exploited and thus disrupt the public digital infrastructure. So far, it has helped uncover over 10,000 security vulnerabilities and 36,000 bugs across more than 1,000 projects, significantly enhancing the quality and security of major software like Chrome, OpenSSL, and systemd.

A project that's part of OSS-Fuzz must have been configured as a ClusterFuzz [42] project. ClusterFuzz is the fuzzing infrastructure that OSS-Fuzz uses under the hood and depends on Google Cloud Platform services, although it can be hosted locally. Such an integration requires setting up a build pipeline, fuzzing jobs and expects a Google Developer account. Results are accessible through a web interface. ClusterFuzz, and by extension OSS-Fuzz, supports fuzzing through LibFuzzer, AFL++, Honggfuzz and FuzzTest—successor to Centipede— with the last two being Google projects [6], [43]–[45]. C, C++, Rust, Go, Python and Java/JVM projects are supported.

3.1.7 OSS-Fuzz-Gen

OSS-Fuzz-Gen (OFG) [28], [46] is Google's current State-Of-The-Art (SOTA) project regarding automatic harness generation through LLMs. It's purpose is to improve the fuzzing infrastructure of open-source projects that are already integrated into OSS-Fuzz. Given such a project, OSS-Fuzz-Gen uses its preexisting fuzzing harnesses and modifies them to produce new ones. Its architecture can be described as follows: 1. With an OSS-Fuzz project's GitHub repository link, OSS-Fuzz-Gen iterates through a set of predefined build templates and generates potential build scripts for the project's harnesses. 2. If any of them succeed they are once again compiled, this time through fuzz-introspector [47]. The latter constitutes a static analysis tool, with fuzzer developers specifically in mind. 3. Build results, old harness and fuzz-introspector report are included in a template-generated prompt, through which an LLM is called to generate a new harness. 4. The newly generated fuzz target is compiled and if it is done so successfully it begins execution inside OSS-Fuzz's infrastructure.

This method proved meaningful, with code coverage in fuzz campaigns increasing thanks to the new generated fuzz drivers. In the case of [48], line coverage went from 38% to 69% without any manual interventions [46].

In 2024, OSS-Fuzz-Gen introduced an experimental feature for generating harnesses in previously unfuzzed projects [49]. The code for this feature resides in the `experimental/from_scratch` directory of the project's GitHub repository [28], with the latest known working commit being `171aac2` and the latest overall commit being four months ago.

3.1.8 AutoGen

AutoGen [26] is a closed-source tool that generates new fuzzing harnesses, given only the library code and documentation. It works as following: The user specifies the function for which a harness is to be generated. AutoGen gathers information for this function—such as the function body, used header files, function calling examples—from the source code and documentation. Through specific prompt templates containing the above information, an LLM is tasked with generating a new fuzz driver, while another is tasked with generating a compilation command for said driver. If the compilation fails, both LLMs are called again to fix the problem, whether it was on the driver's or command's side. This loop iterates until a predefined maximum value or until a fuzz driver is successfully generated and compiled. If the latter is the case, it is then executed. If execution errors exist, the LLM responsible for the driver generation is used to correct them. If not, the pipeline has terminated and a new fuzz driver has been successfully generated.

3.2 Differences

OverHAuL differs, in some way, with each of the aforementioned works. Firstly, although KLEE and IRIS [25], [35] tackle the problem of automated testing and both IRIS and OverHAuL can be considered neurosymbolic AI tools, the similarities end there. None of them utilize LLMs the same way we do—with KLEE not utilizing them by default, as it precedes them chronologically—and neither are automating any part of the fuzzing process.

When it comes to FUDGE, FuzzGen and UTopia [30], [33], [34], all three depend on and demand existing client code and/or unit tests. On the other hand, OverHAuL requires only the bare minimum: the library code itself. Another point of difference is that in contrast with OverHAuL, these tools operate in a linear fashion. No feedback is produced or used in any step and any point failure results in the termination of the entire run.

OverHAuL challenges a common principle of these tools, stated explicitly in FUDGE's paper [34]: "Choosing a suitable fuzz target (still) requires a human". OverHAuL chooses to let the LLM, instead of the user, explore the available functions and choose one to target in its fuzz driver.

OSS-Fuzz-Gen [28] can be considered a close counterpart of OverHAuL, and in some ways it is. A lot of inspiration was gathered from it, like for example the inclusion of static analysis and its usage in informing the LLM. Yet, OSS-Fuzz-Gen has a number of disadvantages that make it in some cases an inferior option. For one, OFG is tightly coupled with the OSS-Fuzz platform [37], which even on its own creates a plethora of issues for the common developer. To integrate their project into OSS-Fuzz, they would need to: Transform it into a ClusterFuzz project [42] and take time to write harnesses for it. Even if these prerequisites are carried out, it probably would not be enough. Per OSS-Fuzz's documentation [39]: "To be accepted to OSS-Fuzz, an open-source project must have a significant user base and/or be critical to the global IT infrastructure". This means that OSS-Fuzz is a viable option only for a small minority of open-source developers and maintainers. One countermeasure of the above shortcoming would be for a developer to run OSS-Fuzz-Gen locally. This unfortunately proves to be an arduous task. As it is not meant to be used standalone, OFG is not packaged in the form of a self-contained application. This makes it hard to setup and difficult to use interactively. Like in the case of FUDGE, OFG's actions are performed linearly. No feedback is utilized nor is there graceful error handling in the case of a step's failure. Even in the case of the experimental feature for bootstrapping unfuzzed projects, OFG's performance varies heavily. During experimentation, a lot of generated harnesses were still wrapped either in Markdown backticks or <code> tags, or were accompanied with explanations inside the generated `.c` source file. Even if code was formatted correctly, in many cases it missed necessary headers for compilation or used undeclared functions.

Lastly, the closest counterpart to OverHAuL is AutoGen [26]. Their similarity stands in the implementation of a feedback loop between LLM and generated harness. However, most other implementation decisions remain distinct. One difference regards the fuzzed function. While AutoGen requires a target function to be specified by the user in which it narrows during its whole run, OverHAuL delegates this to the LLM, letting it explore the codebase and decide by itself the best candidate. Another difference lies in the need—and the lack of—of documentation. While AutoGen requires it to gather information for the given function, OverHAuL leans into the role of a developer by reading the related code and comments and thus avoiding any mismatches between documentation and code. Finally, the LLMs' input is built based on predefined prompt templates, a technique also present in OSS-Fuzz-Gen. OverHAuL operates one abstraction level higher, leveraging DSPy [17] for programming instead of prompting the LLMs used.

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

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

TODO να συμπεριλάβω και τα:

3.2.1 IntelliGen [[20250711141156]]

SAMPLE

IntelliGen: Automatic Fuzz Driver Synthesis Based on Vulnerability Heuristics Zhang et al. (2021) present **IntelliGen**, a system for automatically synthesizing fuzz drivers by statically identifying potentially vulnerable entry-point functions within C projects. Implemented using LLVM, IntelliGen focuses on improving fuzzing efficiency by targeting code more likely to contain memory safety issues, rather than exhaustively fuzzing all available functions.

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

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

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

3.2.2 CKGFuzzer [[20250711203054]]

SAMPLE

CKGFuzzer is a fuzzing framework designed to automate the generation of effective fuzz drivers for C/C++ libraries by leveraging static analysis and large language models. Its workflow begins by parsing the target project along with any associated library APIs to construct a code knowledge graph. This involves two primary steps: first, parsing the abstract syntax tree (AST), and second, performing interprocedural 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. To help determine their severity and root cause, CKGFuzzer consults an LLM-generated knowledge base containing real-world examples of vulnerabilities mapped to known Common Weakness Enumeration (CWE) entries.

3.2.3 PromptFuzz [[20250713225436]]

SAMPLE

Lyu et al. (2024) introduce PromptFuzz [50], a system for automatically generating fuzz drivers using LLMs, with a novel focus on **prompt mutation** to improve coverage. The system is implemented in Rust and targets C libraries, aiming to explore more of the API surface with each iteration.

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

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

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

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

4 Overview

1. How is it different?
2. What does it offer?
3. Example uses
4. Scope of Usage
    1. In what contexts does it work?
    2. Prerequisites

4.1 Architecture

- System diagram
- Main Library Architecture/Structure
- LLM usage
    - Prompting techniques used (callback to Section 2.2.1).
- Static analysis
- Code localization(?)
- Fuzzers
- GitHub Workflow/Usage
- “Ieration budget”

5 Evaluation

5.1 Benchmarks

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

5.2 Performance

5.3 Issues

5.4 Future work

5.4.1 Technical future work

5.4.2 Architectural future work/extensions

1. Build system
2. More (static) analysis tolls integrations
3. General *localization* problem

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 [51]–[54]—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++ [45] could diversify the fuzzing strategies available to OverHAuL, while exploring more experimental tools such as GraphFuzz [29] 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 [7]. 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 [33], [34], this enhancement would enable the tool to exploit programmer-written usage scenarios as seeds or contexts, potentially generating more meaningful and targeted fuzz inputs. Incorporating these richer information sources would likely improve the efficacy of fuzzing campaigns and uncover subtler bugs.

6.2 Experimentation with Large Language Models and Data Representation

OverHAuL's reliance on large language models (LLMs) invites comprehensive experimentation with different providers and architectures to assess their comparative strengths and limitations. Conducting empirical evaluations across leading models—such as OpenAI's o1 and o3 families and Anthropic's Claude Opus 4—will provide valuable insights into their capabilities, cost-efficiency, and suitability for fuzz driver synthesis. Additionally, specialized code-focused LLMs, including generative and fill-in models like Codex-1 and CodeGen [55]–[57], merit exploration due to their targeted optimization for source code generation and understanding.

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

6.3 Comprehensive Evaluation and Benchmarking

To thoroughly establish OverHAuL's effectiveness, extensive large-scale evaluation beyond the initial 10-project corpus is imperative. Applying the tool to repositories indexed in the clib package manager [58], which encompasses hundreds of C libraries, would test scalability and robustness across diverse real-world settings. Such a broad benchmark would also enable systematic comparisons against state-of-the-art automated fuzzing frameworks like OSS-Fuzz-Gen and AutoGen, elucidating OverHAuL's relative strengths and identifying areas for improvement [26], [28].

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

Furthermore, an economic analysis exploring resource consumption—such as API token usage and associated monetary costs—relative to fuzzing effectiveness would be valuable for assessing the practical viability of integrating LLM-based fuzz driver generation into continuous integration processes.

6.4 Practical Deployment and Community Engagement

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

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

7 Discussion

more powerful llms -> better results

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

8 Conclusion

Recap

8.1 Acknowledgements

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

Bibliography

[1] V. J. M. Manes, H. Han, C. Han, *et al.* "The Art, Science, and Engineering of Fuzzing: A Survey." arXiv: 1812.00140 [cs]. (Apr. 7, 2019), [Online]. Available: http://arxiv.org/abs/1812.00140, pre-published.

[2] J. W. Backus, "The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference," in *ICIP Proceedings*, 1959, pp. 125–132. [Online]. Available: https://cir.nii.ac.jp/crid/1572824501224489728.

[3] "Heartbleed Bug." (Mar. 7, 2025), [Online]. Available: https://heartbleed.com/.

[4] C. Meyer and J. Schwenk. "Lessons Learned From Previous SSL/TLS Attacks - A Brief Chronology Of Attacks And Weaknesses." (2013), [Online]. Available: https://eprint.iacr.org/2013/049, pre-published.

[5] "American fuzzy lop." (), [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[6] "libFuzzer – a library for coverage-guided fuzz testing. — LLVM 21.0.0git documentation." (2025), [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

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

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

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

[10] 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.

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

[12] P. Laban, H. Hayashi, Y. Zhou, and J. Neville. "LLMs Get Lost In Multi-Turn Conversation." arXiv: 2505.06120 [cs]. (May 9, 2025), [Online]. Available: http://arxiv.org/abs/2505.06120, pre-published.

[13] N. Perry, M. Srivastava, D. Kumar, and D. Boneh. "Do Users Write More Insecure Code with AI Assistants?" arXiv: 2211.03622. (Dec. 18, 2023), [Online]. Available: http://arxiv.org/abs/2211.03622, pre-published.

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

[15] *Langchain-ai/langgraph*, LangChain, May 21, 2025. [Online]. Available: https://github.com/langchain-ai/langgraph.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] G. Grov, J. Halvorsen, M. W. Eckhoff, B. J. Hansen, M. Eian, and V. Mavroeidis. “On the use of neurosymbolic AI for defending against cyber attacks.” arXiv: 2408.04996. (Aug. 9, 2024), [Online]. Available: http://arxiv.org/abs/2408.04996, pre-published.

[22] 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.

[23] 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.

[24] OpenAI. “ChatGPT.” (2025), [Online]. Available: https://chatgpt.com.

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

[26] 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.

[27] D. Wang, G. Zhou, L. Chen, D. Li, and Y. Miao. “ProphetFuzz: Fully Automated Prediction and Fuzzing of High-Risk Option Combinations with Only Documentation via Large Language Model.” arXiv: 2409.00922 [cs]. (Sep. 1, 2024), [Online]. Available: http://arxiv.org/abs/2409.00922, pre-published.

[28] D. Liu, O. Chang, J. metzman, M. Sablotny, and M. Maruseac, *OSS-fuzz-gen: Automated fuzz target generation*, version https://github.com/google/oss-fuzz-gen/tree/v1.0, May 2024. [Online]. Available: https://github.com/google/oss-fuzz-gen.

[29] H. Green and T. Avgerinos, “GraphFuzz: Library API fuzzing with lifetime-aware dataflow graphs,” in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania: ACM, May 21, 2022, pp. 1070–1081. DOI: 10.1145/3510003.3510228. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510228.

[30] B. Jeong, J. Jang, H. Yi, *et al.*, “UTopia: Automatic Generation of Fuzz Driver using Unit Tests,” in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2676–2692. DOI: 10.1109/SP46215.2023.10179394. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10179394.

[31] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang. “Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT.” arXiv: 2304.02014 [cs]. (Apr. 4, 2023), [Online]. Available: http://arxiv.org/abs/2304.02014, pre-published.

[32] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023, New York, NY, USA: Association for Computing Machinery, Jul. 13, 2023, pp. 423–435, ISBN: 979-8-4007-0221-1. DOI: 10.1145/3597926.3598067. [Online]. Available: https://dl.acm.org/doi/10.1145/3597926.3598067.

[33] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou.

[34] D. Babić, S. Bucur, Y. Chen, *et al.*, "FUDGE: Fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn Estonia: ACM, Aug. 12, 2019, pp. 975–985, ISBN: 978-1-4503-5572-8. DOI: 10.1145/3338906.3340456. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3340456.

[35] 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.

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

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

[38] N. Sasirekha, A. Edwin Robert, and M. Hemalatha, "Program Slicing Techniques and its Applications," *International Journal of Software Engineering & Applications*, vol. 2, no. 3, pp. 50–64, Jul. 31, 2011, ISSN: 09762221. DOI: 10.5121/ijsea.2011.2304. [Online]. Available: http://www.airccse.org/journal/ijsea/papers/0711ijsea04.pdf.

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

[40] CVE Program. "CVE - CVE-2014-0160." (2014), [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

[41] D. Wheeler. "How to Prevent the next Heartbleed." (2014), [Online]. Available: https://dwheeler.com/essays/heartbleed.html.

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

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

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

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

[46] D. Liu, J. Metzman, O. Chang, and G. O. S. S. Team. "AI-Powered Fuzzing: Breaking the Bug Hunting Barrier," Google Online Security Blog. (Aug. 16, 2023), [Online]. Available: https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html.

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

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

[49] OSS-Fuzz Maintainers. "Introducing LLM-based harness synthesis for unfuzzed projects," OSS-Fuzz blog. (May 27, 2024), [Online]. Available: https://blog.oss-fuzz.com/posts/introducing-llm-based-harness-synthesis-for-unfuzzed-projects/.

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

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

[52] S. I. Feldman, "Make — a program for maintaining computer programs," *Software: Practice and Experience*, vol. 9, no. 4, pp. 255–265, 1979, ISSN: 1097-024X. DOI: 10.1002/spe.4380090402. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380090402.

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

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

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

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

[57] OpenAI. “Introducing Codex.” (May 16, 2025), [Online]. Available: https://openai.com/index/introducing-codex/.

[58] “Clib Packages,” GitHub. (2025), [Online]. Available: https://github.com/clibs/clib/wiki/Packages.