/libafl

Source: `~/.claude/skills/tob-testing-handbook-skills/skills/libafl/SKILL.md`

---

name: libafl type: fuzzer description: > LibAFL is a modular fuzzing library for building custom fuzzers. Use for advanced fuzzing needs, custom mutators, or non-standard fuzzing targets.

LibAFL

LibAFL is a modular fuzzing library that implements features from AFL-based fuzzers like AFL++. Unlike traditional fuzzers, LibAFL provides all functionality in a modular and customizable way as a Rust library. It can be used as a drop-in replacement for libFuzzer or as a library to build custom fuzzers from scratch.

When to Use

Fuzzer	Best For	Complexity
libFuzzer	Quick setup, single-threaded	Low

Fuzzer	Best For	Complexity
AFL++	Multi-core, general purpose	Medium
LibAFL	Custom fuzzers, advanced features, research	High

Choose LibAFL when:

- You need custom mutation strategies or feedback mechanisms
- Standard fuzzers don't support your target architecture
- You want to implement novel fuzzing techniques
- You need fine-grained control over fuzzing components
- You're conducting fuzzing research

Quick Start

LibAFL can be used as a drop-in replacement for libFuzzer with minimal setup:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Call your code with fuzzer-provided data
    my_function(data, size);
    return 0;
}
```

Build LibAFL's libFuzzer compatibility layer:

```
git clone https://github.com/AFLplusplus/LibAFL
cd LibAFL/libafl_libfuzzer_runtime
./build.sh
```

Compile and run:

```
clang++ -DNO_MAIN -g -O2 -fsanitize=fuzzer-no-link libFuzzer.a harness.cc main.cc -o fuzz
./fuzz corpus/
```

Installation

Prerequisites

- Clang/LLVM 15-18

- Rust (via rustup)
- Additional system dependencies

Linux/macOS

Install Clang:

```
apt install clang
```

Or install a specific version via apt.llvm.org:

```
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 15
```

Configure environment for Rust:

```
export RUSTFLAGS="-C linker=/usr/bin/clang-15"
export CC="clang-15"
export CXX="clang++-15"
```

Install Rust:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Install additional dependencies:

```
apt install libssl-dev pkg-config
```

For libFuzzer compatibility mode, install nightly Rust:

```
rustup toolchain install nightly --component llvm-tools
```

Verification

Build LibAFL to verify installation:

```
cd LibAFL/libafl_libfuzzer_runtime
./build.sh
# Should produce libFuzzer.a
```

Writing a Harness

LibAFL harnesses follow the same pattern as libFuzzer when using drop-in replacement mode:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // Your fuzzing target code here
    return 0;
}
```

When building custom fuzzers with LibAFL as a Rust library, harness logic is integrated directly into the fuzzer. See the "Writing a Custom Fuzzer" section below for the full pattern.

> **See Also:** For detailed harness writing techniques, see the **harness-writing** technique skill.

Usage Modes

LibAFL supports two primary usage modes:

1. libFuzzer Drop-in Replacement

Use LibAFL as a replacement for libFuzzer with existing harnesses.

Compilation:

```
clang++ -DNO_MAIN -g -O2 -fsanitize=fuzzer-no-link libFuzzer.a harness.cc main.cc -o fuzz
```

Running:

```
./fuzz corpus/
```

Recommended for long campaigns:

```
./fuzz -fork=1 -ignore_crashes=1 corpus/
```

2. Custom Fuzzer as Rust Library

Build a fully customized fuzzer using LibAFL components.

Create project:

```
cargo init --lib my_fuzzer
cd my_fuzzer
cargo add libafl@0.13 libafl_targets@0.13 libafl_bolts@0.13 libafl_cc@0.13 \
  --features "libafl_targets@0.13/libfuzzer,libafl_targets@0.13/sancov_pcguard_hitcounts"
```

Configure Cargo.toml:

```
[lib]
crate-type = ["staticlib"]
```

Writing a Custom Fuzzer

> **See Also:** For detailed harness writing techniques, patterns for handling complex inputs, and advanced strategies, see the **fuzz-harness-writing** technique skill.

Fuzzer Components

A LibAFL fuzzer consists of modular components:

1. **Observers** - Collect execution feedback (coverage, timing)
2. **Feedback** - Determine if inputs are interesting
3. **Objective** - Define fuzzing goals (crashes, timeouts)
4. **State** - Maintain corpus and metadata
5. **Mutators** - Generate new inputs
6. **Scheduler** - Select which inputs to mutate
7. **Executor** - Run the target with inputs

Basic Fuzzer Structure

```
use libafl::prelude::*;
use libafl_bolts::prelude::*;
use libafl_targets::{libfuzzer_test_one_input, std_edges_map_observer};

#[no_mangle]
```

```
pub extern "C" fn libafl_main() {
    let mut run_client = |state: Option<_>, mut restarting_mgr, _core_id| {
        // 1. Setup observers
        let edges_observer = HitcountsMapObserver::new(
            unsafe { std_edges_map_observer("edges") }
        ).track_indices();
        let time_observer = TimeObserver::new("time");

        // 2. Define feedback
        let mut feedback = feedback_or!(
            MaxMapFeedback::new(&edges_observer),
            TimeFeedback::new(&time_observer)
        );

        // 3. Define objective
        let mut objective = feedback_or_fast!(
            CrashFeedback::new(),
            TimeoutFeedback::new()
        );

        // 4. Create or restore state
        let mut state = state.unwrap_or_else(|| {
            StdState::new(
                StdRand::new(),
                InMemoryCorpus::new(),
                OnDiskCorpus::new(&output_dir).unwrap(),
                &mut feedback,
                &mut objective,
            ).unwrap()
        });

        // 5. Setup mutator
        let mutator = StdScheduledMutator::new(havoc_mutations());
        let mut stages = tuple_list!(StdMutationalStage::new(mutator));

        // 6. Setup scheduler
        let scheduler = IndexesLenTimeMinimizerScheduler::new(
            &edges_observer,
            QueueScheduler::new()
```

```
    );

    // 7. Create fuzzer
    let mut fuzzer = StdFuzzer::new(scheduler, feedback, objective);

    // 8. Define harness
    let mut harness = |input: &BytesInput| {
        let buf = input.target_bytes().as_slice();
        libfuzzer_test_one_input(buf);
        ExitKind::Ok
    };

    // 9. Setup executor
    let mut executor = InProcessExecutor::with_timeout(
        &mut harness,
        tuple_list!(edges_observer, time_observer),
        &mut fuzzer,
        &mut state,
        &mut restarting_mgr,
        timeout,
    )?;

    // 10. Load initial inputs
    if state.must_load_initial_inputs() {
        state.load_initial_inputs(
            &mut fuzzer,
            &mut executor,
            &mut restarting_mgr,
            &input_dir
        )?;
    }

    // 11. Start fuzzing
    fuzzer.fuzz_loop(&mut stages, &mut executor, &mut state, &mut restarting_mgr)?;
    Ok(())
};

// Launch fuzzer
Launcher::builder()
```

```
        .run_client(&mut run_client)
        .cores(&cores)
        .build()
        .launch()
        .unwrap();
}
```

Compilation

Verbose Mode

Manually specify all instrumentation flags:

```
clang++-15 -DNO_MAIN -g -O2 \
  -fsanitize-coverage=trace-pc-guard \
  -fsanitize=address \
  -Wl,--whole-archive target/release/libmy_fuzzer.a -Wl,--no-whole-archive \
  main.cc harness.cc -o fuzz
```

Compiler Wrapper (Recommended)

Create a LibAFL compiler wrapper to handle instrumentation automatically.

Create `src/bin/libafl_cc.rs`:

```
use libafl_cc::{ClangWrapper, CompilerWrapper, Configuration, ToolWrapper};

pub fn main() {
    let args: Vec<String> = env::args().collect();
    let mut cc = ClangWrapper::new();
    cc.cpp(is_cpp)
      .parse_args(&args)
      .link_staticlib(&dir, "my_fuzzer")
      .add_args(&Configuration::GenerateCoverageMap.to_flags().unwrap())
      .add_args(&Configuration::AddressSanitizer.to_flags().unwrap())
      .run()
      .unwrap();
}
```

Compile and use:

```
cargo build --release
target/release/libafl_cxx -DNO_MAIN -g -O2 main.cc harness.cc -o fuzz
```

> **See Also:** For detailed sanitizer configuration, common issues, and advanced flags, see the **address-sanitizer** and **undefined-behavior-sanitizer** technique skills.

Running Campaigns

Basic Run

```
./fuzz --cores 0 --input corpus/
```

Multi-Core Fuzzing

```
./fuzz --cores 0,8-15 --input corpus/
```

This runs 9 clients: one on core 0, and 8 on cores 8-15.

With Options

```
./fuzz --cores 0-7 --input corpus/ --output crashes/ --timeout 1000
```

Text User Interface (TUI)

Enable graphical statistics view:

```
./fuzz -tui=1 corpus/
```

Interpreting Output

Output	Meaning
`corpus: N`	Number of interesting test cases found

Output	Meaning
`objectives: N`	Number of crashes/timeouts found
`executions: N`	Total number of target invocations
`exec/sec: N`	Current execution throughput
`edges: X%`	Code coverage percentage
`clients: N`	Number of parallel fuzzing processes

The fuzzer emits two main event types:

- **UserStats** - Regular heartbeat with current statistics
- **Testcase** - New interesting input discovered

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Use `-fork=1 -ignore_crashes=1`	Continue fuzzing after first crash
Use `InMemoryOnDiskCorpus`	Persist corpus across restarts
Enable TUI with `-tui=1`	Better visualization of progress
Use specific LLVM version	Avoid compatibility issues
Set `RUSTFLAGS` correctly	Prevent linking errors

Crash Deduplication

Avoid storing duplicate crashes from the same bug:

Add backtrace observer:

```
let backtrace_observer = BacktraceObserver::owned(
    "BacktraceObserver",
    libafl::observers::HarnessType::InProcess
);
```

Update executor:

```
let mut executor = InProcessExecutor::with_timeout(
    &mut harness,
    tuple_list!(edges_observer, time_observer, backtrace_observer),
    &mut fuzzer,
    &mut state,
    &mut restarting_mgr,
    timeout,
)?;
```

Update objective with hash feedback:

```
let mut objective = feedback_and!(
    feedback_or_fast!(CrashFeedback::new(), TimeoutFeedback::new()),
    NewHashFeedback::new(&backtrace_observer)
);
```

This ensures only crashes with unique backtraces are saved.

Dictionary Fuzzing

Use dictionaries to guide fuzzing toward specific tokens:

Add tokens from file:

```
let mut tokens = Tokens::new();
if let Some(tokenfile) = &tokenfile {
    tokens.add_from_file(tokenfile)?;
}
state.add_metadata(tokens);
```

Update mutator:

```
let mutator = StdScheduledMutator::new(
    havoc_mutations().merge(tokens_mutations())
);
```

Hard-coded tokens example (PNG):

```
state.add_metadata(Tokens::from([
    vec![137, 80, 78, 71, 13, 10, 26, 10], // PNG header
    "IHDR".as_bytes().to_vec(),
```

```
    "IDAT".as_bytes().to_vec(),
    "PLTE".as_bytes().to_vec(),
    "IEND".as_bytes().to_vec(),
]));
```

> **See Also:** For detailed dictionary creation strategies and format-specific dictionaries, see the **fuzzing-dictionaries** technique skill.

Auto Tokens

Automatically extract magic values and checksums from the program:

Enable in compiler wrapper:

```
cc.add_pass(LLVMPasses::AutoTokens)
```

Load auto tokens in fuzzer:

```
tokens += libafl_targets::autotokens()?;
```

Verify tokens section:

```
echo "p (uint8_t *)__token_start" | gdb fuzz
```

Performance Tuning

Setting	Impact
Multi-core fuzzing	Linear speedup with cores
`InMemoryCorpus`	Faster but non-persistent
`InMemoryOnDiskCorpus`	Balanced speed and persistence
Sanitizers	2-5x slowdown, essential for bugs
Optimization level `-O2`	Balance between speed and coverage

Debugging Fuzzer

Run fuzzer in single-process mode for easier debugging:

```
// Replace launcher with direct call
run_client(None, SimpleEventManager::new(monitor), 0).unwrap();

// Comment out:
// Launcher::builder()
//     .run_client(&mut run_client)
//     ...
//     .launch()
```

Then debug with GDB:

```
gdb --args ./fuzz --cores 0 --input corpus/
```

Real-World Examples

Example: libpng

Fuzzing libpng using LibAFL:

1. Get source code:

```
curl -L -O https://downloads.sourceforge.net/project/libpng/libpng16/1.6.37/libpng-
1.6.37.tar.xz
tar xf libpng-1.6.37.tar.xz
cd libpng-1.6.37/
apt install zlib1g-dev
```

2. Set compiler wrapper:

```
export FUZZER_CARGO_DIR="/path/to/libafl/project"
export CC=$FUZZER_CARGO_DIR/target/release/libafl_cc
export CXX=$FUZZER_CARGO_DIR/target/release/libafl_cxx
```

3. Build static library:

```
./configure --enable-shared=no
make
```

4. Get harness:

```
curl -O
https://raw.githubusercontent.com/glennrp/libpng/f8e5fa92b0e37ab597616f554bee254157998227/cont
rib/oss-fuzz/libpng_read_fuzzer.cc
```

5. Link fuzzer:

```
$CXX libpng_read_fuzzer.cc .libs/libpng16.a -lz -o fuzz
```

6. Prepare seeds:

```
mkdir seeds/
curl -o seeds/input.png
https://raw.githubusercontent.com/glennrp/libpng/acfd50ae0ba3198ad734e5d4dec2b05341e50924/cont
rib/pngsuite/iftp1n3p08.png
```

7. Get dictionary (optional):

```
curl -O
https://raw.githubusercontent.com/glennrp/libpng/2fff013a6935967960a5ae626fc21432807933dd/cont
rib/oss-fuzz/png.dict
```

8. Start fuzzing:

```
./fuzz --input seeds/ --cores 0 -x png.dict
```

Example: CMake Project

Integrate LibAFL with CMake build system:

CMakeLists.txt:

```
project(BuggyProgram)
cmake_minimum_required(VERSION 3.0)

add_executable(buggy_program main.cc)

add_executable(fuzz main.cc harness.cc)
target_compile_definitions(fuzz PRIVATE NO_MAIN=1)
target_compile_options(fuzz PRIVATE -g -O2)
```

Build non-instrumented binary:

```
cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ .
cmake --build . --target buggy_program
```

Build fuzzer:

```
export FUZZER_CARGO_DIR="/path/to/libafl/project"
cmake -DCMAKE_C_COMPILER=$FUZZER_CARGO_DIR/target/release/libafl_cc \
      -DCMAKE_CXX_COMPILER=$FUZZER_CARGO_DIR/target/release/libafl_cxx .
cmake --build . --target fuzz
```

Run fuzzing:

```
./fuzz --input seeds/ --cores 0
```

Troubleshooting

Problem	Cause	Solution
No coverage increases	Instrumentation failed	Verify compiler wrapper used, check for `-fsanitize-coverage`
Fuzzer won't start	Empty corpus with no interesting inputs	Provide seed inputs that trigger code paths
Linker errors with `libafl_main`	Runtime not linked	Use `-Wl,--whole-archive` or `-u libafl_main`
LLVM version mismatch	LibAFL requires LLVM 15-18	Install compatible LLVM version, set environment variables
Rust compilation fails	Outdated Rust or Cargo	Update Rust with `rustup update`
Slow fuzzing	Sanitizers enabled	Expected 2-5x slowdown, necessary for finding bugs
Environment variable interference	`CC`, `CXX`, `RUSTFLAGS` set	Unset after building LibAFL project
Cannot attach debugger	Multi-process fuzzing	Run in single-process mode (see Debugging section)

Related Skills

Technique Skills

Skill	Use Case

fuzz-harness-writing	Detailed guidance on writing effective harnesses
address-sanitizer	Memory error detection during fuzzing
undefined-behavior-sanitizer	Undefined behavior detection
coverage-analysis	Measuring and improving code coverage
fuzzing-corpus	Building and managing seed corpora
fuzzing-dictionaries	Creating dictionaries for format-aware fuzzing

Related Fuzzers

Skill	When to Consider
libfuzzer	Simpler setup, don't need LibAFL's advanced features
aflpp	Multi-core fuzzing without custom fuzzer development
cargo-fuzz	Fuzzing Rust projects with less setup

Resources

Official Documentation

- LibAFL Book - Official handbook with comprehensive documentation
- LibAFL GitHub - Source code and examples
- LibAFL API Documentation - Rust API reference

Examples and Tutorials

- LibAFL Examples - Collection of example fuzzers
- cargo-fuzz with LibAFL - Using LibAFL as cargo-fuzz backend
- Testing Handbook LibAFL Examples - Complete working examples from this handbook

---

Revision #5
Created 2026-02-18 08:40:12 UTC by John
Updated 2026-06-21 20:01:26 UTC by John