/aflpp

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

---

name: aflpp type: fuzzer description: > AFL++ is a fork of AFL with better fuzzing performance and advanced features. Use for multi-core fuzzing of C/C++ projects.

AFL++

AFL++ is a fork of the original AFL fuzzer that offers better fuzzing performance and more advanced features while maintaining stability. A major benefit over libFuzzer is that AFL++ has stable support for running fuzzing campaigns on multiple cores, making it ideal for large-scale fuzzing efforts.

When to Use

Fuzzer	Best For	Complexity
AFL++	Multi-core fuzzing, diverse mutations, mature projects	Medium

Fuzzer	Best For	Complexity
libFuzzer	Quick setup, single-threaded, simple harnesses	Low
LibAFL	Custom fuzzers, research, advanced use cases	High

Choose AFL++ when:

- You need multi-core fuzzing to maximize throughput
- Your project can be compiled with Clang or GCC
- You want diverse mutation strategies and mature tooling
- libFuzzer has plateaued and you need more coverage
- You're fuzzing production codebases that benefit from parallel execution

Quick Start

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

Compile and run:

```
# Setup AFL++ wrapper script first (see Installation)
./afl++ docker afl-clang-fast++ -DNO_MAIN=1 -O2 -fsanitize=fuzzer harness.cc main.cc -o fuzz
mkdir seeds && echo "aaaa" > seeds/minimal_seed
./afl++ docker afl-fuzz -i seeds -o out -- ./fuzz
```

Installation

AFL++ has many dependencies including LLVM, Python, and Rust. We recommend using a current Debian or Ubuntu distribution for fuzzing with AFL++.

Method	When to Use	Supported Compilers
Ubuntu/Debian repos	Recent Ubuntu, basic features only	Ubuntu 23.10: Clang 14 & GCC 13 Debian 12: Clang 14 & GCC 12
Docker (from Docker Hub)	Specific AFL++ version, Apple Silicon support	As of 4.35c: Clang 19 & GCC 11

Method	When to Use	Supported Compilers
Docker (from source)	Test unreleased features, apply patches	Configurable in Dockerfile
From source	Avoid Docker, need specific patches	Adjustable via `LLVM_CONFIG` env var

Ubuntu/Debian

Prior to installing afl++, check the clang version dependency of the packge with `apt-cache show afl++`, and install the matching `lld` version (e.g., `lld-17`).

```
apt install afl++ lld-17
```

Docker (from Docker Hub)

```
docker pull aflplusplus/aflplusplus:stable
```

Docker (from source)

```
git clone --depth 1 --branch stable https://github.com/AFLplusplus/AFLplusplus
cd AFLplusplus
docker build -t aflplusplus .
```

From source

Refer to the Dockerfile for Ubuntu version requirements and dependencies. Set `LLVM_CONFIG` to specify Clang version (e.g., `llvm-config-18`).

Wrapper Script Setup

Create a wrapper script to run AFL++ on host or Docker:

```
cat <<'EOF' > ./afl++
#!/bin/sh
AFL_VERSION="${AFL_VERSION:-"stable"}"
case "$1" in
   host)
        shift
        bash -c "$*"
```

```
        ;;
    docker)
        shift
        /usr/bin/env docker run -ti \
            --privileged \
            -v ./:/src \
            --rm \
            --name afl_fuzzing \
            "aflplusplus/aflplusplus:$AFL_VERSION" \
            bash -c "cd /src && bash -c \"$*\""
        ;;
    *)
        echo "Usage: $0 {host|docker}"
        exit 1
        ;;
esac
EOF
chmod +x ./afl++
```

Security Warning: The `afl-system-config` and `afl-persistent-config` scripts require root privileges and disable OS security features. Do not fuzz on production systems or your development environment. Use a dedicated VM instead.

System Configuration

Run after each reboot for up to 15% more executions per second:

```
./afl++ <host/docker> afl-system-config
```

For maximum performance, disable kernel security mitigations (requires grub bootloader, not supported in Docker):

```
./afl++ host afl-persistent-config
update-grub
reboot
./afl++ <host/docker> afl-system-config
```

Verify with `cat /proc/cmdline` - output should include `mitigations=off`.

Writing a Harness

Harness Structure

AFL++ supports libFuzzer-style harnesses:

```
#include <stdint.h>
#include <stddef.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    // 1. Validate input size if needed
    if (size < MIN_SIZE || size > MAX_SIZE) return 0;

    // 2. Call target function with fuzz data
    target_function(data, size);

    // 3. Return 0 (non-zero reserved for future use)
    return 0;
}
```

Harness Rules

Do	Don't
Reset global state between runs	Rely on state from previous runs
Handle edge cases gracefully	Exit on invalid input
Keep harness deterministic	Use random number generators
Free allocated memory	Create memory leaks
Validate input sizes	Process unbounded input

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

Compilation

AFL++ offers multiple compilation modes with different trade-offs.

Compilation Mode Decision Tree

Choose your compilation mode:

- **LTO mode** (`afl-clang-lto`): Best performance and instrumentation. Try this first.
- **LLVM mode** (`afl-clang-fast`): Fall back if LTO fails to compile.
- **GCC plugin** (`afl-gcc-fast`): For projects requiring GCC.

Basic Compilation (LLVM mode)

```
./afl++ <host/docker> afl-clang-fast++ -DNO_MAIN=1 -O2 -fsanitize=fuzzer harness.cc main.cc -o fuzz
```

GCC Compilation

```
./afl++ <host/docker> afl-g++-fast -DNO_MAIN=1 -O2 -fsanitize=fuzzer harness.cc main.cc -o fuzz
```

Important: GCC version must match the version used to compile the AFL++ GCC plugin.

With Sanitizers

```
./afl++ <host/docker> AFL_USE_ASAN=1 afl-clang-fast++ -DNO_MAIN=1 -O2 -fsanitize=fuzzer harness.cc main.cc -o fuzz
```

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

Build Flags

Note that `-g` is not necessary, it is added by default by the AFL++ compilers.

Flag	Purpose
`-DNO_MAIN=1`	Skip main function when using libFuzzer harness

Flag	Purpose
`-O2`	Production optimization level (recommended for fuzzing)
`-fsanitize=fuzzer`	Enable libFuzzer compatibility mode and adds the fuzzer runtime when linking executable
`-fsanitize=fuzzer-no-link`	Instrument without linking fuzzer runtime (for static libraries and object files)

Corpus Management

Creating Initial Corpus

AFL++ requires at least one non-empty seed file:

```
mkdir seeds
echo "aaaa" > seeds/minimal_seed
```

For real projects, gather representative inputs:

- Download example files for the format you're fuzzing
- Extract test cases from the project's test suite
- Use minimal valid inputs for your file format

Corpus Minimization

After a campaign, minimize the corpus to keep only unique coverage:

```
./afl++ <host/docker> afl-cmin -i out/default/queue -o minimized_corpus -- ./fuzz
```

> **See Also:** For corpus creation strategies, dictionaries, and seed selection, see the **fuzzing-corpus** technique skill.

Running Campaigns

Basic Run

```
./afl++ <host/docker> afl-fuzz -i seeds -o out -- ./fuzz
```

Setting Environment Variables

```
./afl++ <host/docker> AFL_FAST_CAL=1 afl-fuzz -i seeds -o out -- ./fuzz
```

Interpreting Output

The AFL++ UI shows real-time fuzzing statistics:

Output	Meaning
execs/sec	Execution speed - higher is better
cycles done	Number of queue passes completed
corpus count	Number of unique test cases in queue
saved crashes	Number of unique crashes found
stability	% of stable edges (should be near 100%)

Output Directory Structure

```
out/default/
├── cmdline          # How was the SUT invoked?
├── crashes/         # Inputs that crash the SUT
│   └── id:000000,sig:06,src:000002,time:286,execs:13105,op:havoc,rep:4
├── hangs/           # Inputs that hang the SUT
├── queue/           # Test cases reproducing final fuzzer state
│   ├── id:000000,time:0,execs:0,orig:minimal_seed
│   └── id:000001,src:000000,time:0,execs:8,op:havoc,rep:6,+cov
├── fuzzer_stats     # Campaign statistics
└── plot_data        # Data for plotting
```

Analyzing Results

View live campaign statistics:

```
./afl++ <host/docker> afl-whatsup out
```

Create coverage plots:

```
apt install gnuplot
./afl++ <host/docker> afl-plot out/default out_graph/
```

Re-executing Test Cases

```
./afl++ <host/docker> ./fuzz out/default/crashes/<test_case>
```

Fuzzer Options

Option	Purpose
`-G 4000`	Maximum test input length (default: 1048576 bytes)
`-t 1000`	Timeout in milliseconds for each test case (default: 1000ms)
`-m 1000`	Memory limit in megabytes (default: 0 = unlimited)
`-x ./dict.dict`	Use dictionary file to guide mutations

Multi-Core Fuzzing

AFL++ excels at multi-core fuzzing with two major advantages:

1. More executions per second (scales linearly with physical cores)
2. Asymmetrical fuzzing (e.g., one ASan job, rest without sanitizers)

Starting a Campaign

Start the primary fuzzer (in background):

```
./afl++ <host/docker> afl-fuzz -M primary -i seeds -o state -- ./fuzz 1>primary.log
2>primary.error &
```

Start secondary fuzzers (as many as you have cores):

```
./afl++ <host/docker> afl-fuzz -S secondary01 -i seeds -o state -- ./fuzz 1>secondary01.log
2>secondary01.error &
./afl++ <host/docker> afl-fuzz -S secondary02 -i seeds -o state -- ./fuzz 1>secondary02.log
2>secondary02.error &
```

Monitoring Multi-Core Campaigns

List all running jobs:

```
jobs
```

View live statistics (updates every second):

```
./afl++ <host/docker> watch -n1 --color afl-whatsup state/
```

Stopping All Fuzzers

```
kill $(jobs -p)
```

Coverage Analysis

AFL++ automatically tracks coverage through edge instrumentation. Coverage information is stored in `fuzzer_stats` and `plot_data`.

Measuring Coverage

Use `afl-plot` to visualize coverage over time:

```
./afl++ <host/docker> afl-plot out/default out_graph/
```

Improving Coverage

- Use dictionaries for format-aware fuzzing
- Run longer campaigns (cycles_wo_finds indicates plateau)
- Try different mutation strategies with multi-core fuzzing
- Analyze coverage gaps and add targeted seed inputs

> **See Also:** For detailed coverage analysis techniques, identifying coverage gaps, and systematic coverage improvement, see the **coverage-analysis** technique skill.

CMPLOG

CMPLOG/RedQueen is the best path constraint solving mechanism available in any fuzzer. To enable it, the fuzz target needs to be instrumented for it. Before building the fuzzing target set the environment variable:

```
./afl++ <host/docker> AFL_LLVM_CMPLOG=1 make
```

No special action is needed for compiling and linking the harness.

To run a fuzzer instance with a CMPLOG instrumented fuzzing target, add `-c0` to the command like arguments:

```
./afl++ <host/docker> afl-fuzz -c0 -S cmplog -i seeds -o state -- ./fuzz 1>secondary02.log
2>secondary02.error &
```

Sanitizer Integration

Sanitizers are essential for finding memory corruption bugs that don't cause immediate crashes.

AddressSanitizer (ASan)

```
./afl++ <host/docker> AFL_USE_ASAN=1 afl-clang-fast++ -DNO_MAIN=1 -O2 -fsanitize=fuzzer
harness.cc main.cc -o fuzz
```

Note: Memory limit (`-m`) is not supported with ASan due to 20TB virtual memory reservation.

UndefinedBehaviorSanitizer (UBSan)

```
./afl++ <host/docker> AFL_USE_UBSAN=1 afl-clang-fast++ -DNO_MAIN=1 -O2 -
fsanitize=fuzzer,undefined harness.cc main.cc -o fuzz
```

Common Sanitizer Issues

Issue	Solution
ASan slows fuzzing	Use only 1 ASan job in multi-core setup
Stack exhaustion	Increase stack with `ASAN_OPTIONS=stack_size=...`

Issue	Solution
GCC version mismatch	Ensure system GCC matches AFL++ plugin version

> **See Also:** For comprehensive sanitizer configuration and troubleshooting, see the **address-sanitizer** technique skill.

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Use LLVMFuzzerTestOneInput harnesses where possible	If a fuzzing campaign has at least 85% stability then this is the most efficient fuzzing style. If not then try standard input or file input fuzzing
Use dictionaries	Helps fuzzer discover format-specific keywords and magic bytes
Set realistic timeouts	Prevents false positives from system load
Limit input size	Larger inputs don't necessarily explore more space
Monitor stability	Low stability indicates non-deterministic behavior

Standard Input Fuzzing

AFL++ can fuzz programs reading from stdin without a libFuzzer harness:

```
./afl++ <host/docker> afl-clang-fast++ -O2 main_stdin.c -o fuzz_stdin
./afl++ <host/docker> afl-fuzz -i seeds -o out -- ./fuzz_stdin
```

This is slower than persistent mode but requires no harness code.

File Input Fuzzing

For programs that read files, use `@@` placeholder:

```
./afl++ <host/docker> afl-clang-fast++ -O2 main_file.c -o fuzz_file
./afl++ <host/docker> afl-fuzz -i seeds -o out -- ./fuzz_file @@
```

For better performance, use `fmemopen` to create file descriptors from memory.

Argument Fuzzing

Fuzz command-line arguments using `argv-fuzz-inl.h`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef __AFL_COMPILER
#include "argv-fuzz-inl.h"
#endif

void check_buf(char *buf, size_t buf_len) {
    if(buf_len > 0 && buf[0] == 'a') {
        if(buf_len > 1 && buf[1] == 'b') {
            if(buf_len > 2 && buf[2] == 'c') {
                abort();
            }
        }
    }
}

int main(int argc, char *argv[]) {
#ifdef __AFL_COMPILER
    AFL_INIT_ARGV();
#endif

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <input_string>\n", argv[0]);
        return 1;
    }

    char *input_buf = argv[1];
    size_t len = strlen(input_buf);
    check_buf(input_buf, len);
    return 0;
}
```

Download the header:

```
curl -O
https://raw.githubusercontent.com/AFLplusplus/AFLplusplus/stable/utils/argv_fuzzing/argv-fuzz-inl.h
```

Compile and run:

```
./afl++ <host/docker> afl-clang-fast++ -O2 main_arg.c -o fuzz_arg
./afl++ <host/docker> afl-fuzz -i seeds -o out -- ./fuzz_arg
```

Performance Tuning

Setting	Impact
CPU core count	Linear scaling with physical cores
Persistent mode	10-20x faster than fork server
`-G` input size limit	Smaller = faster, but may miss bugs
ASan ratio	1 ASan job per 4-8 non-ASan jobs

Real-World Examples

Example: libpng

Fuzzing libpng demonstrates fuzzing a C project with static libraries:

```
# Get source
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/

# Install dependencies
apt install zlib1g-dev

# Configure and build static library
export CC=afl-clang-fast CFLAGS=-fsanitize=fuzzer-no-link
export CXX=afl-clang-fast++ CXXFLAGS="$CFLAGS"
```

```
./configure --enable-shared=no
export AFL_LLVM_CMPLOG=1
export AFL_USE_ASAN=1
make

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

# Link fuzzer
export AFL_USE_ASAN=1
$CXX -fsanitize=fuzzer libpng_read_fuzzer.cc .libs/libpng16.a -lz -o fuzz

# Prepare seeds and dictionary
mkdir seeds/
curl -o seeds/input.png
https://raw.githubusercontent.com/glennrp/libpng/acfd50ae0ba3198ad734e5d4dec2b05341e50924/cont
rib/pngsuite/iftp1n3p08.png
curl -O
https://raw.githubusercontent.com/glennrp/libpng/2fff013a6935967960a5ae626fc21432807933dd/cont
rib/oss-fuzz/png.dict

# Start fuzzing
./afl++ <host/docker> afl-fuzz -i seeds -o out -- ./fuzz
```

Example: CMake-based Project

```
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 -O2 -fsanitize=fuzzer-no-link)
target_link_libraries(fuzz -fsanitize=fuzzer)
```

Build and fuzz:

```
# Build non-instrumented binary
./afl++ <host/docker> cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ .
./afl++ <host/docker> cmake --build . --target buggy_program

# Build fuzzer
./afl++ <host/docker> cmake -DCMAKE_C_COMPILER=afl-clang-fast -DCMAKE_CXX_COMPILER=afl-clang-
fast++ .
./afl++ <host/docker> cmake --build . --target fuzz

# Fuzz
./afl++ <host/docker> afl-fuzz -i seeds -o out -- ./fuzz
```

Troubleshooting

Problem	Cause	Solution
Low exec/sec (<1k)	Not using persistent mode	Create a LLVMFuzzerTestOneInput style harness
Low stability (<85%)	Non-deterministic code	Fuzz a program via stdin or file inputs, or create such a harness
GCC plugin error	GCC version mismatch	Ensure system GCC matches AFL++ build and install gcc-$GCC_VERSION-plugin-dev
No crashes found	Need sanitizers	Recompile with `AFL_USE_ASAN=1`
Memory limit exceeded	ASan uses 20TB virtual	Remove `-m` flag when using ASan
Docker performance loss	Virtualization overhead	Use bare metal or VM for production fuzzing

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	Detect undefined behavior bugs

Skill	Use Case
fuzzing-corpus	Building and managing seed corpora
fuzzing-dictionaries	Creating dictionaries for format-aware fuzzing

Related Fuzzers

Skill	When to Consider
libfuzzer	Quick prototyping, single-threaded fuzzing is sufficient
libafl	Need custom mutators or research-grade features

Resources

Key External Resources

AFL++ GitHub Repository Official repository with comprehensive documentation, examples, and issue tracker.

Fuzzing in Depth Advanced documentation by the AFL++ team covering instrumentation modes, optimization techniques, and advanced use cases.

AFL++ Under The Hood Technical deep-dive into AFL++ internals, mutation strategies, and coverage tracking mechanisms.

AFL++: Combining Incremental Steps of Fuzzing Research Research paper describing AFL++ architecture and performance improvements over original AFL.

Video Resources

- Fuzzing cURL - Trail of Bits blog post on using AFL++ argument fuzzing for cURL
- Sudo Vulnerability Walkthrough - LiveOverflow series on rediscovering CVE-2021-3156
- Rediscovery of libpng bug - LiveOverflow video on finding CVE-2023-4863

Revision #5
Created 2026-02-18 08:40:10 UTC by John
Updated 2026-06-21 20:01:20 UTC by John