# [AFL - American Fuzzy Lop](http://lcamtuf.coredump.cx/afl/)

American fuzzy lop is a security-oriented [fuzzer](https://en.wikipedia.org/wiki/Fuzzing) that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. The genetic algorithms aim to efficiently increase code coverage of the test cases. It helps testing software to find unexpected results within applications.

\*\*Testing Approach:\*\* Fuzz testing

\*\*Testing Level:\*\* Application testing

\*\*Target application domain:\*\* Application system

## Features

\* \*\*Fuzzing engine\*\*: The fuzzing engine of american fuzzy lop uses several algorithms whose goal is to trigger unexpected behavior, including bit flips or replacing bytes of input file with various integers that can trigger edge cases. Apart from that, it can generate test cases based on sample keywords, which helps during fuzzing of programs that employed text-based grammar, such as SQLite. Generated test cases that exercise different parts of the program's code can later be used as input for more specialized diagnostic programs. While performing the fuzzing, a hang can be detected when the process does not exit within the specified timeout and crash is assumed when a signal handler kills the process. The fuzzed input can be fed to the tested program either via standard input or as an input file specified in the process command line. Fuzzing networked programs is currently not directly supported, although in some cases there are feasible solutions to this problem.

\* \*\*Performance\*\*: One of the challenges american fuzzy lop had to solve involved an efficient spawning of hundreds of processes per second. Apart from the original engine that spawned every process from scratch, american fuzzy lop offers the default engine that relies heavily on fork system call. This can further be sped up by leveraging LLVM deferred forkserver mode, but this comes at the cost of having to modify the tested program. Also, american fuzzy lop supports fuzzing the same program over the network.

\* \*\*User Interface\*\*: American fuzzy lop features a colorful command line interface that displays real-time statistics about the fuzzing process. Various settings may be triggered by either command line options or environment variables. Apart from that, programs may read runtime statistics from files in a machine-readable format.

\* \*\*Utility programs\*\*: In addition to afl-fuzz and tools that can be used for binary instrumentation, american fuzzy lop features utility programs meant for monitoring of the fuzzing process. Apart from that, there is afl-cmin and afl-tmin, which can be used for test case and test corpus minimization. This can be useful when the test cases generated by afl-fuzz would be used by other fuzzers.

## Pros

\* Designed to be practical

\* It has modest performance overhead

\* Uses a variety of highly effective fuzzing strategies and effort minimization tricks

\* Real-world use cases

\* It is pretty sophisticated

\* It has street smarts

\* It is fast

\* No tinkering required

\* It's chainable to other tools

\* It sports a hip, retro-style UI

## Cons

\* Finds only crashes (typical of most fuzzers)

\* File input only

\* Linux/OSX only

\* Need to build target from sourse

\* Gets stuck on magic values

\* No native parallelisation

## Required information / models

The program requires the user to provide a sample command that runs the tested application and at least one small example input file.

After this initial phase, AFL begins the actual process of fuzzing by applying various modifications to the input file. When the tested program crashes or hangs, this might suggest the discovery of a new bug, possibly a security vulnerability. In this case, the modified input file is saved for further user inspection.

In order to maximize the fuzzing performance, american fuzzy lop expects the tested program to be compiled with the aid of a utility program that instruments the code with helper functions which track control flow. This allows the fuzzer to detect when the target's behavior changes in response to the input. In cases when this is not possible, black-box testing is supported as well.

## Target platform and dependencies

The afl tool is written in C.

The tool is confirmed to work on x86 Linux, OpenBSD, FreeBSD, and NetBSD, both 32- and 64-bit. It should also work on MacOS X and Solaris, although with some constraints. It supports programs written in C, C++, or Objective C, compiled with either gcc or clang. On Linux, the optional QEMU mode allows black-box binaries to be fuzzed, too.

There are variants and derivatives of AFL that allow to fuzz Python, Go, Rust, OCaml, GCJ Java, kernel syscalls, or even entire VMs. There is also a closely inspired in-process fuzzer baked into LLVM and a fork that runs on Windows. Finally, AFL is one of the fuzzing engines behind OSS-Fuzz.

\*\*Dependencies:\*\*

\* Linux machine

\* Basic C and command line experience

## Updates

For four years there was strong contribution activity but, the last commit is almost one and half year ago. Due to the fact that there are multiple uses of the specific tool I suppose that the tool reached in a very stable version.

\*\*Latest update:\*\* 2017-11-04

\*\*First release date:\*\* 2013-11-12

## Licensing / Cost

Apache License 2.0 / Free

## Tutorials and documentation

\* Tutorial: There is an afl-training available [here](https://github.com/ThalesIgnite/afl-training). The specific training contains extra material like presentation, video, link for afl-users mailing list, link for workshop and tutorial.

\* Documentation: The official documentation is available [here](http://lcamtuf.coredump.cx/afl/QuickStartGuide.txt)

## Usage examples

There are multiple examples of tools that used the afl, as they are presented in the official afl site. Some instances are:

\* [iOS Kernel](https://support.apple.com/en-us/HT208331): about a security update

\* [Mozilla Firefox](https://lcamtuf.blogspot.com/2014/09/cve-2014-1564-uninitialized-memory-when.html): about an image parsing issue

\* [GnuPG](https://blog.fuzzing-project.org/2-Buffer-overflow-and-other-minor-issues-in-GnuPG-and-libksba-TFPA-0012014.html): about a buffer overflow issue

\* [Android](https://www.blackhat.com/docs/us-15/materials/us-15-Drake-Stagefright-Scary-Code-In-The-Heart-Of-Android.pdf): about an issue in the media server

## Alternative tools

\* [0d1n](https://github.com/CoolerVoid/0d1n/)

\* [Granota](https://github.com/valldrac/granota)

\* [boofuzz](https://github.com/jtpereyda/boofuzz)

\* [dirsearch](https://github.com/maurosoria/dirsearch)