Differential Fuzzing on coreutils Using LibAFL

—

Report

Valentin Huber

Institute of Applied Information Technology
Zürich University of Applied Sciences ZHAW
contact@valentinhuber.me

May 28, 2024

Abstract

Hello, World!

Contents

1 Introduction

1.1 Differential Fuzzing

- Fuzzing has been popular and effective at finding bugs.
- Much research has gone into input selection/guiding.
- Not much has been done when it comes to oracles.
- Differential Fuzzing: What it is.

1.2 coreutils

- Short overview
- uutils' version

1.3 LibAFL

- History of AFL/AFL++
- Problems with incompatible forks and how LibAFL attempts to fix them

1.4 Research Questions

1. Which parts of coreutils can be fuzzed? What performance tradeoffs does each part introduce?
2. How can the necessary instrumentation be introduced into coreutils? What are the engineering and performance implications of each option?
3. Can LibAFL feasibly be used to build a system with all logic defined in the answers to the questions above?
4. If yes, how effective is the resulting fuzzer at finding bugs in coreutils? What kind of bug can be found with it?
5. Can the system be expanded to implement differential fuzzing between the different implementations? What changes are necessary?
6. If yes, how effective is the this second fuzzer at finding bugs in coreutils? What kind of bug can be found with it?

2 Background

2.1 coreutils

On Feb. 8, 1990, D. J. MacKenzie announced fileutils, a suite of utilities for reading and altering files [1]. A year later, he released textutils (to parse and manipulate text) [2] and shellutils (to write powerful shell scripts) [3]. These three collections were folded into one on Jan. 13, 2003, called the GNU coreutils. [4] `ls`, `cat`, `base64`, `grep`, `env`, or `whoami`: GNU's coreutils are at the basis of how users interact with most Linux distributions on the command line. [5] Because they are so widely used and central to how users interact with their computers, software quality and lack of software defects is especially important to coreutils.

Version 9.5 of the GNU coreutils was released on Mar. 28, 2024 and thus marks the current version as of this report. 106 programs are built per default. [6]

2.1.1 Interface

Users primarily interact with coreutils through the command line or in shell scripts. They take different kinds of inputs, i.e. behave differently based on changes to:

- Data passed to `stdin`, e.g. through Unix pipes
- Command line arguments:
  - Unnamed arguments, either required (such as `cp <source> <destination>`) or optional (such as `ls [directory]`)
  - Flags without any associated data, such as `--help`
  - Flags with associated data, either required (such as `dd if=<input file> of=<output file>`) or optional such as `-name <pattern>` in `find`
- Environment variables, such as `LANG`
- The file system content, such as for `ls`

The output, or effects of invocations fall into the following categories:

- Data written to `stdout`
- Data written to `stderr`
- The exit status of the process
- The signal terminating the process
- Changes to the file system

2.1.2 Alternative Implementations

Since the release of GNU coreutils, multiple alternative implementations were released. Notable among those are BusyBox [7], which aims to provide most of the GNU coreutils with a focus on resource restrictions. It is therefore primarily used in embedded systems [7] or tiny distributions such as Alpine Linux [8].

In the general push towards rewriting software in memory-safe languages, the uutils project [9] maintains a drop-in replacement implementation of the GNU coreutils written in Rust. [10] It does contain all programs, but is still missing certain options. All differences with GNU's coreutils are treated as bugs. It further aims to not only work on Linux, but is also available for MacOS and Windows.

2.1.3 Build System

The GNU coreutils employ a complex, multi-step build system, including Autoconf [11] and Automake [12]. Changes to either the code or the build system configuration require a deep understanding of the entire ecosystem to ensure no unintended changes to the resulting binaries are introduced. uutils' version of coreutils relies on cargo as its build system, which make outcomes of changes to the code much more predictable.

2.2 LibAFL

As introduced in Section 1.3, LibAFL is an extendible framework to build custom fuzzers. What follows is a short introduction to the ideas and parts of LibAFL necessary to understand this project. For a more in-depth look, the authors refer to the original LibAFL paper [13].

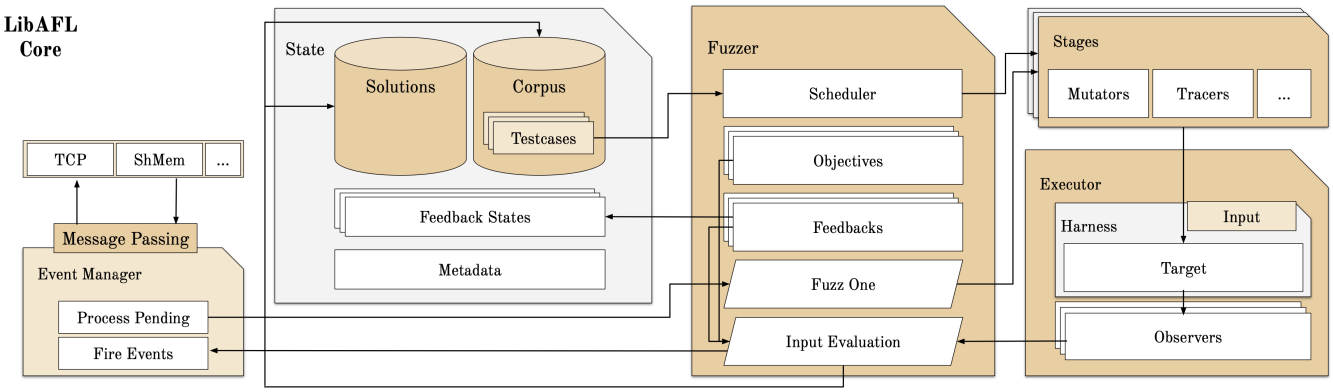

FIGURE 1: LibAFL's Architecture [13]

2.2.1 Generic Concepts

The core insight by the authors of LibAFL is that most fuzzers contain very similar parts, where advancements in new works are usually developments in only few of the fundamental fuzzer parts. The logic is usually either introduced into a fork of an existing fuzzer (such as AFL++), which is then abandoned without collecting the advancements in a way that they are reusable and combinable in later projects. Alternatively, authors might elect to create a new system from scratch, which introduces a lot of duplicate work and code, which isn't as optimized as possible, since fuzzer parts such as the thread synchronization usually aren't the focus of new works.

The authors of LibAFL examined recent innovations in fuzzing and identified a set of distinct parts present in many of the fuzzers. They then designed the architecture of LibAFL in such a way that authors can change just the parts they are interested in and rely on a well-optimized default implementation for everything else. Figure1 shows the identified fuzzer parts and their interaction, as implemented in LibAFL. Using this architecture, the authors of LibAFL re-implemented the logic introduced in high-impact 20 works to both proof that the system is flexible enough to handle logic created indepen from their system. Finally, they showed that it is now trivial to combine advancements made in different works (i.e. combining the input scheduler of one work with new feedback types introduced in another).

2.2.2 Usage of Traits

On a more technical side, to achieve their goal, the authors heavily relied on Rust's trait system in combination with generic types. This section will present a selection of these core traits, including a short explanation. For a complete list and more expansive documentation, the author refers to the paper which introduced LibAFL [13] and the official LibAFL book [14].

Executor The `Executor` is called with a certain `Input` and executes the program under test. It ensures that the passed `Observer`s can gather whatever information they need to and provide additional information about the execution run. Examples for `Executors` include `InProcessExecutor`, which calls the provided function directly, `InProcessForkExecutor`, which calls `fork` before calling the function and trades off additional performance overhead for improved stability. Finally, while being libraries external to the LibAFL core, `FridaInProcessExecutor` and `QemuExecutor` use the dynamic instrumentation framework Frida resp. the emulator QEMU to support complex fuzzing approaches.

Observer `Observer`s are passed to an `Executor` and gather information before, during, or after the execution. This information can take any arbitrary shape. Typical `Observer` implementations used in fuzzers include `StdMapObserver`, which is often used to keep track of coverage data by recording execution paths, or `CmpObserver` to trace comparisons during an execution run.

Feedback There are two uses for `Feedback`s in LibAFL: objectives and feedbacks steering the search. They reduce a combination of information provided by the `Executor` and `Observer`s to a boolean value: Is this test case interesting? If used as an objective, this marks the test case in question as a solution. If used as a steering feedback, it will add the current test case to the corpus from which (after mutating it) the next input is drawn. Typical examples include a `CrashFeedback`, `TimeoutFeedback` (both usually used in the objective mode) and a `MaxMapFeedback`, which can be used to see if new parts of the memory map used to keep track of the executed parts of the binary are touched, thus representing the coverage-guided part of the fuzzer.

Input `Input`s are about what one would expect: A representation of what is mutated and then passed to the `Executor`. Typically, it consists of a simple variable length byte array, but may also be a more complex `struct`. Depending on the use case, the former might be sufficient or one might need the additional flexibility provided by the latter.

Mutator Finally, `Mutator`s take an `Input` and change it in some form. On simple byte array `Input`s, this may be flipping a bit, or injecting an additional byte somewhere. For more complex `Input`s, custom `Mutator`s may need to be written. All `Mutator`s included in a certain fuzzer are passed to a scheduler which mutates an `Input` once or multiple times with the received `Mutator`s. This is another example of the flexibility of LibAFL: While one may need to write custom `Mutator`s for a custom `Input` type, all scheduling algorithms available in LibAFL can be used regardless, without any additional changes.

3 Implementation

This section introduces the concepts and technical details necessary to reproduce the findings of this work by incrementally adding capabilities to a fuzzer. The example fuzzer will target `base64`, a comparatively small program from the coreutils. It encodes and decodes binary data to a format consisting only of characters unproblematic in most contexts. It takes its input from either a file or `stdin` and provides the following options:

- `--decode` switches `base64`'s mode from encode to decode
- `--ignore-garbage` ignores non-alphabet characters when decoding
- `--wrap <cols>` inserts linebreak after `<cols>` characters when encoding
- `--help` prints information about `base64` and exits
- `--version` prints version information and exits

3.1 Basic Unguided Fuzzer

The first step is to build a fuzzer without any features: It just takes a byte array, randomly mutates it, feeds it to the fuzzer and checks for crashes. It does not contain any execution steering nor can it trigger the different options. One can choose the default implementations for each part outlined in Section2.2, which reduces the necessary code for the fuzzer to well under 100 lines. However, this fuzzer is very unlikely to find

any software defect, since it essentially employs the same strategy as was proposed in the seminal work by Miller *et al.* in 1990: inserting random data into a program and hoping for a crash, without any additional logic. Additionally, it does not even have access to all parts of the program, since the command line arguments are never used.

3.1.1 Environment Protection

The most simple solution to allow a fuzzer to access the command line arguments would be to split the byte array at a magic byte (e.g. `NULL` bytes), and pass the first entry to `stdin` and all remaining as command line arguments. However, this introduces a problem: Some programs in the coreutils can change the environment they are running in, as described in Section 2.1.1. This may be entirely trivial (e.g. creating and writing to a temporary file in an unrelated part of the filesystem), but may also disturb the fuzzing process or even incur irreparable damage to the system by overwriting critical files.

The fuzzer therefore needs to protect the environment from the effects of the program. This can be done in a few different ways, each of which introduces a certain downside:

1. First, the fuzzer could create a layered filesystem and use it to create an environment for the program under test to run in. This is what Docker uses to use the host's kernel while fooling the program under test into thinking it runs natively. Changes to the file system are captured and stored while read operations are responded to with data from the write layer if it has been changed previously and from the host's file system if not. However, the performance implications of this approach are immense: While starting a coreutils program takes about 20 ms on the author's system, starting a Docker container takes approximately 2 s. Additionally, doing this across many cores, as is typical in modern fuzzing, relies on the parallelization of starting containers as done by the Docker daemon. This is not necessarily the case, in fact in earlier work by the author [16], sub-linear scaling effects could be observed.

2. Alternatively, a dynamic translation layer could be introduced that captures the relevant syscalls and handles them appropriately. While this would limit the startup overhead, it would slow down the program execution. Additionally, the logic necessary in the dynamic environment protection layer is non-trivial and may depend on both the program under test and the specific system used to run the fuzzer.

3. Many programs in the coreutils don't change the environment they run in, or only do so for very specific options. So while unable to reach all code, restricting the fuzzing campaigns to the program parts which do not change their environment would prevent any performance overhead at the cost of completeness. This is the approach pursued in this project.

3.1.2 Custom Input Type

This approach requires restricting the fuzzer to only execute whitelisted parts of the program under test, specifically to only add certain command line arguments. This also means that certain parts of the command line argument parsing routines will never get tested, since only valid command line arguments will be tested. The associated data to a certain flag (see Section 2.1.1) may still be invalid and the corresponding parsing routine will be tested.

Since this project uses LibAFL, this can be achieved quite easily by implementing a custom `Input` type (refer to Section 2.2.2). By introducing a trait which contains functions that map the `Input` to the arguments necessary for the `Executor`, it is possible to build a system where the only addition to the codebase for additional programs to test is adding

- a custom `Input` struct,
- a mapping function for the `Executor`,
- a few simple trait implementations needed for the fuzzer (such as `Display`), with many necessary implementations available as `derive`s,
- a `Generator` for the above, which will generate random seeds for the fuzzer to start from, and
- a set of `Mutators`, which will mutate the parts of the `Input` independently. For this part, the author introduced a system which allows reusing the default byte array mutators for any `Input` part consisting of a byte array.

3.2 Optimizations

This basic fuzzer can then be augmented by systems that LibAFL provide. With very little additional code, the fuzzer can be extended to run on all available cores or even multiple machines, use advanced algorithms to choose the best `Mutator`, etc. Additional `Observers` and `Feedbacks`, such as

a `TimeoutFeedback` can be introduced with no additional configuration. This is where LibAFL as a framework simplifies building an advanced fuzzer significantly.

3.3 Gathering Coverage Information from GNU coreutils

To test any non surface level code, the fuzzer needs information of some form about what parts of the binary just got executed. This coverage information can fundamentally be gathered in two ways: Either the necessary logic is compiled into the binary, or it is added dynamically. While the former is more performant, it also requires changes to the binary. And as described in Section 2.1.3, making changes to the code or build system of GNU coreutils is a complex task. Previous experiments by the author on coreutils showed that in principle, adding compile-time coverage-gathering instrumentation is possible. [16]

3.3.1 LibAFL's Coverage Interface

LibAFL heavily relies on shared memory maps for a wide range of internal functionality like corpus synchronization across threads. It is further important for different kinds of `Feedback`, especially coverage information. Its built-in logic for adding coverage gathering instrumentation to a binary to test relies on passes in the clang compiler, specifically the `SanitizerCoverage` [17] module. This module includes different levels of coverage instrumentation, the examples provided by LibAFL typically use `trace-pc-guard`. This will insert the following call at every edge:

```
1 __sanitizer_cov_trace_pc_guard(&
    ↪ guard_variable)
2
```

The implementation of this function is then left for the developer. The LibAFL module `libafl_targets` provides such implementations which allocate a shared memory map with the correct size and then on the execution of each edge marks the memory section associated with it. Finally, a `MaxMapFeedback` is used as feedback in the fuzzer, which makes the fuzzer prioritize inputs that visited new paths in the binary under test, since additional bits are set in the shared memory map.

However, these default implementations only work if the fuzzer and binary under test exist in the same process, i.e. when the fuzzer and source code to test are compiled into a single unit. Based on the reasoning in Section 2.1.3, this is not a feasible solution for this project. Hence, a different approach was created.

3.3.2 Instrumentation

First, inspired by the simple default implementation provided in the documentation for `SanitizerCoverage` [17], a simple implementation for the required function is written where the map created by the pass in the tested binary is marked as the edges are executed. Then, additional exported functions are written which make the gathered information available to other parts of the binary. This file is then compiled to an object file.

In a next step, the GNU coreutils are built using the following flags:

- For the compiler (`CFLAGS`):
  - `-g` retains the symbol information in the compiled binary.
  - `-fsanitize-coverage=trace-pc-guard` introduces the function calls as specified above. Note: The custom implementation is not linked to it yet, it only contains a weakly linked default implementation.
- For the linker (`LDFLAGS`):
  - `-rdynamic` adds the code's symbols to the dynamic linking table to be available in dynamically linked binaries.
  - `$(realpath ./coverage.o)` includes the previously compiled object file in the linker sources. The linker will then override the weakly linked default implementation with the custom implementation found in this binary.

These steps produce binaries which behave exactly as produced by an unmodified compilation process, but have additional functionality statically compiled in, which records coverage information and makes it available through functions available in dynamically linked binaries.

Bibliography

[1] D. J. MacKenzie, *Gnu file utilities release 1.0*, Email to gnu.utils.bug Email List, Feb. 8, 1990. [Online]. Available: https://groups.google.com/g/gnu.utils.bug/c/CviP42X_hCY/m/YssXFn-JrX4J (visited on May 22, 2024).

[2] D. J. MacKenzie, *New gnu file and text utilities released*, Email to gnu.utils.bug Email List, Jul. 15, 1991. [Online]. Available: https://groups.google.com/g/gnu.utils.bug/c/iN5KuoJYRhU/m/V_6oiBAWF0EJ (visited on May 22, 2024).

[3] D. J. MacKenzie, *Gnu shell programming utilities released*, Email to gnu.utils.bug Email List, Aug. 22, 1991. [Online]. Available: https://groups.google.com/g/gnu.utils.bug/c/xpTRtuFpNQc/m/mRc_7JWZ0BYJ (visited on May 22, 2024).

[4] J. Meyering, *Package renamed to coreutils*, git commit, Jan. 13, 2003. [Online]. Available: https://git.savannah.gnu.org/cgit/coreutils.git/tree/README-package-renamed-to-coreutils (visited on May 22, 2024).

[5] R. Stallman. "Linux and the gnu system." (Nov. 2, 2021), [Online]. Available: https://www.gnu.org/gnu/linux-and-gnu.en.html (visited on May 22, 2024).

[6] J. Meyering, P. Brady, B. Voelker, E. Blake, P. Eggert, and A. Gordon, *Gnu coreutils 9.5*, Software Release, Mar. 28, 2024. [Online]. Available: https://ftp.gnu.org/gnu/coreutils/coreutils-9.5.tar.gz (visited on May 22, 2024).

[7] "Busybox: The swiss army knife of embedded linux." (), [Online]. Available: https://www.busybox.net/about.html (visited on May 22, 2024).

[8] "Alpine linux — about." (), [Online]. Available: https://alpinelinux.org/about/ (visited on May 22, 2024).

[9] "Uutils." (), [Online]. Available: https://uutils.github.io/ (visited on May 22, 2024).

[10] "Uutils — coreutils." (), [Online]. Available: https://uutils.github.io/coreutils (visited on May 22, 2024).

[11] J. Meyering. "Autoconf." (Dec. 8, 2020), [Online]. Available: https://www.gnu.org/software/autoconf (visited on May 24, 2024).

[12] J. Meyering. "Automake." (Jan. 31, 2022), [Online]. Available: https://www.gnu.org/software/automake (visited on May 24, 2024).

[13] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1051–1065, ISBN: 9781450394505. DOI: 10.1145/3548606.3560602. [Online]. Available: https://doi.org/10.1145/3548606.3560602.

[14] A. Fioraldi and D. Maier. "The libafl fuzzing library." (), [Online]. Available: https://aflplus.plus/libafl-book/ (visited on May 28, 2024).

[15] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279.

[16] V. Huber, "Running klee on gnu coreutils," Feb. 2024. [Online]. Available: https://github.com/riesentoaster/klee-coreutils-experiments/releases/download/v1.0/Huber-Valentin-running-KLEE-on-coreutils-report.pdf.

[17] "Sanitizercoverage." (), [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html (visited on May 28, 2024).

[18] A. Sjöbom and A. Hasselberg. "Fuzzing." (Apr. 25, 2019), [Online]. Available: https://github.com/adamhass/fuzzing/ (visited on May 21, 2024).

[19] M. Moroz. "Fuzzing." (Jun. 8, 2021), [Online]. Available: https://github.com/google/AFL (visited on May 21, 2024).