Differential Fuzzing on coreutils Using LibAFL

—

Report

Valentin Huber

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

June 4, 2024

Abstract

Hello, World!

Contents

1 Introduction

Fuzz testing has developed into an important tool for finding software defects. By repeatedly running a program under test with varied input data and detecting illegal states such as crashes, it is an automated alternative to manual security and reliability testing. Introduced in 1990 in the seminal work by Miller *et al.* [1], it is now widely used in industry. Companies such as Google, Microsoft, Cisco, or Adobe and governmental agencies such as the US Department of Defense have developed proprietary fuzzers or contributed to open source fuzzers. Fuzz testing has proven an effective tool for both security and reliability testing. Its accolades include, among others, 20,000 discovered vulnerabilities in Google's Chrome browser alone. [2]

1.1 Differential Fuzzing

While extensive research has been conducting on fuzzing, one field which has not received a lot of attention is the oracle which defines what an illegal state of the program under test constitutes. While simple oracles such as program crashes and timeouts are easy to detect, further exploration has not gone past very general logic, such as memory-violation bugs. [3]

One such oracle is what is used in so-called differential fuzzing. It relies on two independent implementations of the same underlying program logic and works by comparing their output under the same input. Compared to other common and more general oracles, this allows detecting logic errors that do not invalidate basic guardrails such as memory access rules.

1.2 coreutils

This work will examine coreutils as an example of a target to perform differential fuzzing on. coreutils are a suite of programs that allow users to interact with their system on the command line. Popularized by the version developed by the GNU project and available on almost all current Linux systems, utilities such as `ls`, `cat`, `base64`, `grep`, `env`, or `whoami` are indispensible tools for many users' daily workflows. Section 2.1 will provide a more in-depth introduction to coreutils along with information about its history, interface and technical build-up.

1.3 LibAFL

American Fuzzy Lop (AFL) [4] was one early comprehensive open-source fuzzing tool used in countless projects and academic works to detect an impressive list of software defects and security vulnerabilities [5]. After it was no longer updated in November 2017, the fork AFL++ [6] has become the de facto replacement.

However, while advancements on AFL++ have been introduced in many academic and commercial projects, because of the architecture of AFL++, these usually have not been introduced back to AFL++. This results in a list of incompatible forks, each with a proven improvement that makes the fuzzer more effective at its job. [7] Because of this, the maintainers of AFL++ started a new project: LibAFL. It aims to provide a toolkit for building fuzzers that is flexible enough to allow combination of all these advancements. Refer to Section 2.2 for more details on LibAFL. This work uses LibAFL to build a fuzzer aimed to find software defects in coreutils.

1.4 Research Questions

The remainder of this work aims to answer the following 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?

Refer to Section 6.1 for a summary of the answers produced in this work.

2 Background

To understand the requirements to and the architecture of the fuzzer, some background information on coreutils and LibAFL's architecture are necessary. This section provides an overview of these topics.

2.1 coreutils

On Feb. 8, 1990, D. J. MacKenzie announced fileutils, a suite of utilities for reading and altering files [8]. A year later, he released textutils (to parse and manipulate text) [9] and shellutils (to write powerful shell scripts) [10]. These three collections were folded into one on Jan. 13, 2003, called the GNU coreutils. [11] `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. [12] 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. [13]

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 [14], which aims to provide most of the GNU coreutils with a focus on resource restrictions. It is therefore primarily used in embedded systems [14] or tiny distributions such as Alpine Linux [15].

In the general push towards rewriting software in memory-safe languages, the uutils project [16] maintains a drop-in replacement implementation of the GNU coreutils written in Rust. [17] 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 [18] and Automake [19]. 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 [7].

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.

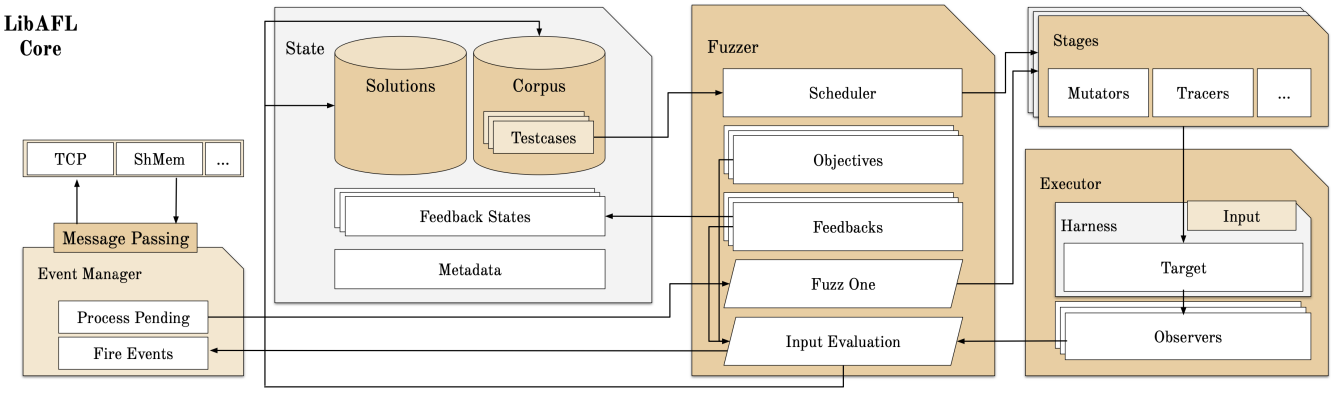

FIGURE 1: LibAFL's Architecture [7]

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 [7] and the official LibAFL book [20].

Executor The `Executor` is called with a certain `Input` and executes the program under test. It ensures that the passed `Observers` 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 `Observers` 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 `Feedbacks` in LibAFL: objectives and feedbacks steering the search. They reduce a combination of information provided by the `Executor` and `Observers` 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 `Inputs` 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 [21], 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 over-

head 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 `Mutator`s, 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 `Observer`s and `Feedback`s, 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. [21]

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` [22] module. This module includes different levels of coverage instrumentation, the examples provided by LibAFL typically use `trace-pc-guard`. This will insert the call shown in Listing 1 on every edge.

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

Listing 1: Inserted Call on Every Edge

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` [22], 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. `realpath` has to be included since `make` will traverse subdirectories where the relative path is no longer correct.

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.

3.3.3 Dynamic Interface

In a next step, this functionality needs to be made accessible to the fuzzer. This is accomplished by building dynamic system libraries, which are passed to the loader using the `LD_PRELOAD` environment variable. They can hook into the execution process at multiple points to execute their logic. And, because the symbolic information is kept in the binaries under test, they can call the functions providing coverage data described in Section 3.3.2. In this project, two binaries are employed

Coverage Map Size Extraction The first of these hooks into `libc_start_main`, and instead of calling the binaries main function calculates the size of the memory map the coverage data is stored in and prints it to `stdout`. This can then be read by the fuzzer to allocate its own memory map of the same size. This binary is used only once at the beginning of a fuzzing campaign, since the map size does not change.

Coverage Extraction The second binary performs more complex logic:

1. Before the binary is ever called, the fuzzer allocates a shared memory map with the size as extracted by the method described above. The memory map needs to be available to child processes spawned by the fuzzer. To achieve this, `mmap` and `shm_open` based shared memory is used. Additionally, the `libc` flag `FD_CLOEXEC` needs to be unset to ensure access in the child process.
2. A LibAFL `ShmemDescription` of this memory map is then created. For `mmap` style memory maps, this contains a file descriptor and the size of the map. The descriptor is then serialized to a printable format.
3. The fuzzer only then calls the binary under test, with the second dynamic system library loaded using again `LD_PRELOAD`. Besides all arguments used in the fuzzing run, as a last argument it passes the serialized shared memory description.
4. The helper binary then deserializes the description and makes it available for later use.
5. It also changes `argc` and `argv` to remove the description.
6. With the changed arguments, it then calls `libc_start_main` of the binary to test.
7. The helper binary also adds a hook to one of the teardown functions of the process. Once this is called, it retrieves the access to the shared memory prepared as described above. It then uses one of the functions defined in the custom coverage information collection code to retrieve the coverage data from the current execution and copies it to the shared memory.
8. Once the process is done, the fuzzer continues with its analysis. And transparent to the rest of the fuzzer, as if done by the default modules in `libafl_targets`, the coverage information can be found in the shared memory. It can then be analyzed with all the default `Observer`s and `Feedback`s.

This entire process is transparent to the binary under test and the entire fuzzer except for the `Executor` and

some code to set up the shared memory. This makes it very flexible to use, even in cases like this project, where more intrusive changes to the program under test are not feasible.

3.4 Gathering Coverage Information from uutils coreutils

The approach to gathering coverage information from uutils' version of coreutils is similar to the one described above. Since Rust uses clang under the hood as well, the challenge is reduced to passing the compiler and linker flags in the correct way. Because direct access to `CLFAGS` and `LDFLAGS` is limited, only `CFLAGS="-g"` is passed this way. The remaining functionality is passed using `RUSTFLAGS`:

- `-Cpasses=sancov-module` is needed to enable the `SanitizerCoverage` pass
- `-Cllvm-args=-sanitizer-coverage-trace-pc-guard` sets the correct mode
- `-Cllvm-args=-sanitizer-coverage-level=3` has to be set manually here, where it is set to `3` automatically if clang is invoked manually
- `-Clink-arg=-rdynamic` again keeps the symbol information in the final binary
- `-Clink-arg=$(realpath ./coverage.o)` passes the coverage file

The remaining steps including the helper binaries is the same as described with GNU's coreutils.

3.5 Differential Fuzzing

As introduced in Section 1.1, differential fuzzing describes an advancement in oracle development. It executes different implementations of the same logic and compares their outputs. Typical oracles define fixed logic to determine if a certain execution should be considered a solution. Typical oracles for this are simple but unspecific (i.e. producing false negatives), like checks for crashes or timeouts. Simple heuristic checks for logic consistency between input and output of a program under test suffer from either a significant false negative or false positive rate.
To check the full logic, one would essentially need a second reference implementation of the program under test. The fuzzer can then simply check the outputs of the programs for differences. While this obviously cannot find an error if it is present in all implementations, many bugs can still be detected.
However, creating such reference implementations is not feasible for fuzzing alone. Therefore, differential fuzzing is almost exclusively performed when multiple implementations of the same logic are already present. Section 5 presents relevant such works.

3.5.1 Existing Functionality and Custom Extensions

LibAFL includes certain existing functionality for differential fuzzing, including

- `DiffExecutor`, which takes two other `Executors` which are then used to perform the actual executions and whose `ExitKinds` (very coarse exit reason distinction, specifically `Ok`, `Crash`, or `Timeout`) are compared
- `DifferentialObserver`, a trait for `Observers` to pass to the `DiffExecutor`, including some implementations, such as for `StdMapObserver`, and
- a very simple example fuzzer performing differential fuzzing and showing the usage of the above.

Based on the already existing code, only two additional `Feedbacks` were required:

- `DiffExitKindFeedback` checks if the `ExitKinds` of the two `Executors` in the `DiffExecutor` were different and builds on functionality in `DiffExecutor`. This addition was submitted to the upstream project by the author and has already been accepted. Refer to Section 6.2 for a complete list of the contributions to LibAFL based on this project.
- `DiffStdIOMetadataPseudoFeedback` is necessary to extract vital information about differences in `stdout` and `stderr` between the two binaries tested into the logged error case.

4 Results

Unfortunately, LibAFL's binary data `Mutators`' interface made it so that only one part of the `Input` could be altered by them. While a lengthy discussion about this limitation lead to significant improvements including a clear path to a more flexible solution, the necessary changes were not introduced in time to be considered in this project. Because of this, the evaluation of the produced fuzzer can only be done in a rather limited fashion.
As an example of a program from the different implementations of coreutils that only consists of one

binary `Input` part, `base64` was selected as an evaluation target. Refer to Section 3 for an explanation of its functionality.

4.1 Different Behavior Declared Consistent

uutils' website claims that "Differences with GNU are treated as bugs" [17]. However, this seems to only hold for information written to `stdout`. If an invalid command is issued, the version of the program under test by uutils returns additional information, such as a usage tip. Listings 2 and 3 show the differences in error message when passing an illegal argument.

```
$ gnu-base64 --invalid-flag
gnu-base64: unrecognized option '--invalid-
    ↪ flag'
Try 'gnu-base64 --help' for more information.
```

Listing 2: Error Message When Passing an Illegal Flag to GNU's Version of Coreutils

```
$ uutils-base64 --invalid-flag
error: unexpected argument '--invalid-flag'
    ↪ found

  tip: to pass '--invalid-flag' as a value,
    ↪ use '-- --invalid-flag'

Usage: uutils-base64 [OPTION]... [FILE]

For more information, try '--help'.
```

Listing 3: Error Message When Passing an Illegal Flag to uutils's Version of Coreutils

Because of this, in its differential mode, the fuzzer reduces the information from the `stderr` of each process to a check for *any* output. It reports an error, if either one implementation returns any data on `stderr` and the other does not, or if both do not print any data to `stderr`, but print different data to `stdout`.
The latter restriction is required because of implementation details: In `base64`, GNU's version prints valid bytes during the decoding process and, once it encounters invalid data, prints an error message. uutils' version on the other hand first parses the entire input and only then prints the output or error message. An example output can be seen in Listings 4 and 5.

```
$ echo "aa" | gnu-base64 --decode
ignu-base64: invalid input
```

Listing 4: Error Message Including Partial Output (leading `i`) from GNU's `base64`

```
$ echo "aa" | uutils-base64 --decode
uutils-base64: error: invalid input
```

Listing 5: Error Message Without Partial Output from uutils' `base64`

Finally, the option printing the version (aptly named `--version`) obviously prints different information. Similarly, `--help` prints different text, since it includes the file path. These options are unlikely to introduce business logic error because of their simplicity and are thus both disregarded in this project.

4.2 Performance

Across 64 fully loaded cores of an AMD EPYC 7713 with 64 GB of RAM, the complete fuzzer in its differential fuzzing modes achieves approximately 1500 executions per second. According to the fuzzer, only 3.405% of edge coverage is reached.
To gather additional information about this number, the GNU coreutils were additionally compiled with instrumentation for coverage analysis with `gcov`. The fuzzer was then instructed to execute `Input`s, that reached new edges on this version of the program under test. This is accomplished by introducing a new `Feedback`, which can be attached to the `MaxMapFeedback` using boolean logic operators in a way it is only evaluated when the `MaxMapFeedback` considers the `Input` interesting. This new `Feedback` returns a constant interestingness and simply runs the `gcov` instrumented binary with the current `Input`. This results in only 1690 executions on an evaluation run with millions of executions on the binaries under test, thus introducing negligible performance overhead.
Analysis of the results reported by `gcov` shows that 62.26% of 212 lines were executed. Manual inspection of the coverage file showed that the missing lines either consist of `stdio` error handling code (e.g. write operations to `stdout`, which should never fail, since pipes aren't intentionally broken by the fuzzer) and code for the `--version` and `--help` flags.

4.2.1 Advanced LibAFL Modules

Section 3.2 introduced how LibAFL simplifies introducing advanced algorithms for `Input` for `Mutator` selection. Figure 2 shows how introducing two of these changes the performance of the fuzzer. For this experiment, each configuration is run ten times until the maximum achievable coverage is reached and the runtime according to LibAFL is recorded. On average, it took the fuzzer 95.5 s to explore all reachable edges.

In a second step, a `StdWeightedScheduler` is introduced. This is LibAFL's implementation of the algorithm introduced by AFLFast [23] and explores low-frequency paths using a Markov chain model using a power schedule based on energy calculated for each input. In comparison, the `StdScheduler` picks a random `Input` from the corpus to mutate next. Introducing this improvement effectively reduces variability and the average runtime to 74.3 s, mostly by eliminating outliers with a high runtime.
Finally, the basic `StdScheduledMutator` is replaced by `StdMOptMutator`. While the former picks a random number of random mutators to call, the latter calculates the mutator most likely to find improvements by employing particle swarm optimizations, as introduced in MOPT [24]. This introduces a calculation overhead which outweighs gains by improved `Mutator` selection, as visible in Figure 2. This tradeoff may change with increased program length and complexity. The execution speed reported by the fuzzer approached and equaled the same level, regardless of the specific optimization level used, after a few minutes.

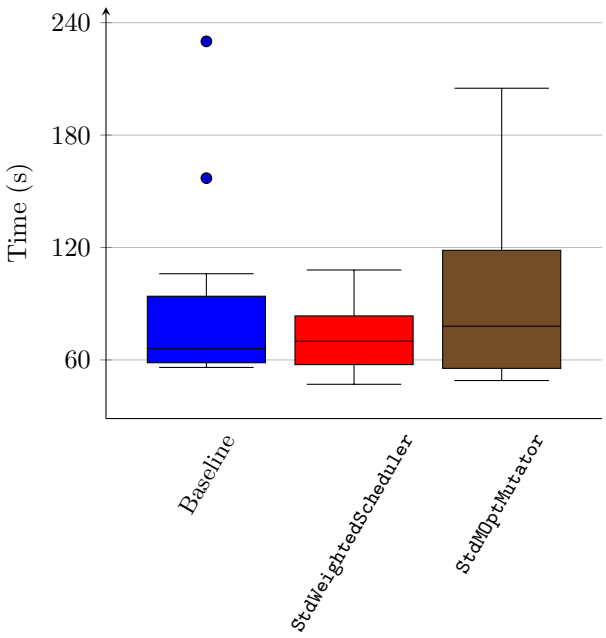

FIGURE 2: Time to Reach the Maximum Coverage on `base64` Across Advanced LibAFL Modules

4.3 Performance on Individual Implementations

4.4 Discovered Errors

5 State of the Art

The author could not find any previous project attempting to employ differential fuzzing to find software errors in coreutils. However, there exist projects fuzzing coreutils using other approaches and projects employing differential fuzzing on other targets. This section presents a selection of these.

5.1 Fuzzing coreutils

The seminal work introducing fuzz testing to the scientific community tested a suite of utilities on different UNIX systems including some that would later be found in coreutils. Their system was very simple: Generate random data (either completely random, random data without `NULL` bytes, or random printable characters) and check if the utility crashes. By doing this, they found errors in between a quarter and a third of utilities, depending on the UNIX system they were testing. [1]
The authors later repeated their experiments, most recently in 2022. However, they still followed a black-box fuzzing approach with no execution guidance or other techniques developed in the past 30 years. They also tested uutils' implementation and found similar levels of error prevalence. [25]

5.1.1 Symbolic Execution Frameworks

KLEE is a symbolic execution based whitebox fuzzer that relies on SMT solvers to calculate inputs that will cover new paths in a program. It showed that this approach can be highly effective at achieving high coverage in both the GNU and BusyBox versions of coreutils. [26] It also forms the base of a list of fuzzers that build on top of KLEE to introduce new concepts. Examples include:

- CRETE introduced the concept of concolic execution: Running the program under test with concrete values and tracing the instructions executed to then calculate new inputs based on this instead of keeping the entire state-space in memory. [27]
- FUSE only selectively performs symbolic execution based on function complexity analysis. [28]
- Learch introduced a state selection algorithm

that learns which state contributes most towards maximizing coverage. [29]

- Cloud9 proposed a custom state-merging algorithm to help mitigate the state-explosion problem inherent in symbolic execution. [30]

5.1.2 Other approaches

Outside of symbolic execution based fuzzing, coreutils have received little attention in academia. Here is what the author was able to find:

- CLIFuzzer [31] automatically extracts information about the command line interface from the program in question and then generates inputs based on this grammar. However, it does not take coverage information or any other guidance heuristic into account.

- AFL included an argv fuzzing mode, however, according to the author, "it's just not horribly useful in practice" [32]. Sjöbom and Hasselberg attempted to evaluate its effectiveness, but showed the drawbacks outlined in Section 3.1.1. [33]

5.1.3 Other Related Works

Finally, certain loosely fuzzing related projects use coreutils in some part of their evaluation:

- SEDiff performs differential fuzzing on coreutils, but is not looking at software errors in the programs itself but instead evaluates function summaries used in symbolic execution based fuzzing. [34]

- Kilmer evaluates the top vulnerability discovery tool from the DARPA Cyber Grand Challenge on more complex programs that are closer to what is expected in real-world software. They use coreutils to evaluate prevalence of C library functions in real-world software. [35]

- IMF-SIM is an attempt to de-obfuscate programs by comparing a vector calculated based on the behavior of them. This vector is generated by a fuzzing-like process. The authors evaluate their approach on coreutils. [36]

- Building seccomp filters for Linux sandboxing manually is a tedious and error-prone task. Gelderie. *et al.* employs a fuzzer-like program to automatically generate all possible syscalls from a program. [37]

5.2 Differential Fuzzing

6 Discussion

This work presents a new fuzzer built on LibAFL, designed to perform differential fuzzing between different implementations of coreutils. Even though, because of current limitations in LibAFL, only a single program could be tested, it can be shown that this approach is promising in finding logic errors even in simple programs such as `base64`. This section discusses the findings in this work, its contributions and limitations, and attempts to answer the research questions posed at the beginning.

6.1 Research Questions

Section 1.4 introduced a series of questions to be answered in this work. Here is a summary of what was found.

In principle, everything in coreutils can be tested by a fuzzer. However, based on the interface information in Section 2.1.1 and the considerations on environment protection in Section 3.1.1, a fuzzer whose goal complete coverage is needs to introduce a computationally expensive layer to record environmental changes and prevent their persistence across executions of the program under test. Additionally, a layer to emulate failing system interactions (such as broken pipes) needs to be introduces. The performance cost of these layers make them infeasible to use in testing without large compute resources. Additionally, as was shown in Section 4.4, even a fuzzer without these capabilities can find software defects and is thus worth pursuing.

This work further introduces a novel approach to LibAFL to gather coverage information from a binary called in a child process. Sections 3.3 and 3.4 describe how edge coverage is gathered by logic statically compiled into the binary under test and extracted by dynamically loaded libraries, transparent to the program under test.

The fuzzer built in this project makes extensive use of LibAFL's features, including idiomatic usage of LibAFL's internal representation of coverage information (see Section 3.3.1), usage of advanced algorithms as provided by LibAFL (see Section 3.2 and 4.2.1), and experiments on custom `Input` types (as described in Section 3.1.2 and discussed in Section 6.3.2).

A non-differential fuzzer can only find bugs that trigger states that are easily distinguishable from normal operation. Section 3.5 explains why this in practice reduces the discoverable errors to those that either crash the program under test or produce timeouts.

Section 4.3 shows that even long evaluation runs on `base64` did not discover any such bugs.
Section 3.5.1 lists existing functionality in LibAFL used in the fuzzer in its differential mode, along with custom extensions introduced during this project. Because of LibAFL's modular design, running two `Executor`s in parallel, including all their respective `Observer`s, and introducing `Feedback`s comparing their output is not only possible but becomes an engineering task of limited difficulty.
Finally, the fuzzer in its differential mode finds non-trivial logic errors in the different implementations, undiscoverable by its non-differential mode. It explores all program parts except for those excluded based on environmental protection reasons as described above.

6.2 Contributions

During this project, several improvements to LibAFL's upstream repository have been introduced:

- `CommandExecutor` was generalized to work on custom `Input`s, rather than simple byte array `Input`s only (PR #2129)
- Improvements to `StdOutObserver` and `StdErrObserver` (PR #2236)
- Introduction of `DiffExitKindFeedback`, which compares the `ExitKind` of two `Executor`s in differential fuzzing and reports discrepancies (PR #2246)

Based on feedback by the author of this work, the following changes were introduced by LibAFL's maintainers:

- Architectural improvements on handling of `StdOutObserver` and `StdErrObserver`
- Work on `tuple_list` and moving from `HasBytesArray` to `HasMutatorBytes`, in preparation of introducing `MappingMutator` to allow byte array mutation based on default `Mutator`s on arbitrary parts of a custom `Input`

Further improvements were discussed but could not yet be introduced because of time limitations, but may be in the coming months:

- `MappingMutator` as described above, including an example fuzzer showing handling of custom `Input`s
- A `CommandExecutor` implementation for custom `Input`s, where the mapping of `Input` parts to the different parts of the command is defined on the `Input` based on a trait
- A set of macros to automatically generate the mutators necessary for a custom `Input`
- An example fuzzer including possible further additions to LibAFL's source code to show how coverage information can be retrieved when using `CommandExecutor`, as done in this project

6.3 Limitations and Future Work

While the fuzzer as introduced in the previous sections was already able to produce results in the form of prove-of-concept work and found bugs, it still has some limitations.

6.3.1 Untested Program Parts

Sections 2.1.1 and 6.1 discuss how the fuzzer is limited to functionality that does not change the environment. Additionally, code handling errors in the programs input and output (e.g. broken pipes) is not tested because of a missing intermediate layer artificially introducing these errors. Future projects may introduce these missing layers either individually or combined to test the code sections not tested by this project.

6.3.2 Limitations on Custom Input Type Mutations

Section 4 explains why the evaluation of the fuzzer was only performed on `base64`. After the missing additions described in Section 6.2, specifically `MappingMutator` to mutate multiple byte array parts of a custom `Input` type, are introduced, a more complete evaluation will be possible.

6.4 Summary

Bibliography

[1] 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.

[2] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, Oct. 2023, ISSN: 0360-0300. DOI: 10.1145/3623375. [Online]. Available: https://doi.org/10.1145/3623375.

[3] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, Sep. 2022, ISSN: 0360-0300. DOI: 10.1145/3512345. [Online]. Available: https://doi.org/10.1145/3512345.

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

[5] M. Zalewski. "The bug-o-rama trophy case." (), [Online]. Available: https://lcamtuf.coredump.cx/afl/#bugs (visited on Jun. 3, 2024).

[6] "AFL++." (), [Online]. Available: https://aflplus.plus (visited on Jun. 3, 2024).

[7] 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.

[8] 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).

[9] 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).

[10] 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).

[11] 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).

[12] 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).

[13] 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).

[14] "BusyBox: The Swiss Army Knife of Embedded Linux." (), [Online]. Available: https://www.busybox.net/about.html (visited on May 22, 2024).

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

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

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

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

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

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

[21] 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.

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

[23] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019. DOI: 10.1109/TSE.2017.2785841.

[24] C. Lyu, S. Ji, C. Zhang, *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[25] B. P. Miller, M. Zhang, and E. R. Heymann, "The relevance of classic fuzz testing: Have we solved this one?" *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2028–2039, 2022. DOI: 10.1109/TSE.2020.3047766.

[26] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[27] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "Crete: A versatile binary-level concolic testing framework," in *Fundamental Approaches to Software Engineering*, A. Russo and A. Schürr, Eds., Cham: Springer International Publishing, 2018, pp. 281–298, ISBN: 978-3-319-89363-1.

[28] G. Zhang, Z. Chen, Z. Shuai, Y. Zhang, and J. Wang, "Synergizing symbolic execution and fuzzing by function-level selective symbolization," in *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, 2022, pp. 328–337. DOI: 10.1109/APSEC57359.2022.00045.

[29] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2526–2540, ISBN: 9781450384544. DOI: 10.1145/3460120.3484813. [Online]. Available: https://doi.org/10.1145/3460120.3484813.

[30] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, 2012, pp. 193–204, ISBN: 9781450312059. DOI: 10.1145/2254064.2254088. [Online]. Available: https://doi.org/10.1145/2254064.2254088.

[31] A. Gupta, R. Gopinath, and A. Zeller, "Clifuzzer: Mining grammars for command-line invocations," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, ¡conf-loc¿, ¡city¿Singapore¡/city¿, ¡country¿Singapore¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2022, pp. 1667–1671, ISBN: 9781450394130. DOI: 10.1145/3540250.3558918. [Online]. Available: https://doi.org/10.1145/3540250.3558918.

[32] M. Zalewski. "Struggling to give inputs to AFL." (), [Online]. Available: https://groups.google.com/g/afl-users/c/ZBWq0LdHBzw/m/zBlo7q9LBAAJ (visited on Jun. 4, 2024).

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

[34] P. Li, W. Meng, and K. Lu, "Sediff: Scope-aware differential fuzzing to test internal function models in symbolic execution," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, ¡conf-loc¿, ¡city¿Singapore¡/city¿, ¡country¿Singapore¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2022, pp. 57–69, ISBN: 9781450394130. DOI: 10.1145/3540250.3549080. [Online]. Available: https://doi.org/10.1145/3540250.3549080.

[35] E. D. Kilmer, "Extending vulnerability discovery with fuzzing and symbolic execution to realistic applications," M.S. thesis, Penn State University, Jun. 9, 2017.

[36] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 319–330. DOI: 10.1109/ASE.2017.8115645.

[37] M. Gelderie., V. Barth., M. Luff., and J. Birami., "Seccomp filters from fuzzing," in *Proceedings of the 19th International Conference on Security and Cryptography - SECRYPT*, INSTICC, SciTePress, 2022, pp. 507–512, ISBN: 978-989-758-590-6. DOI: 10.5220/0011145100003283.