Differential Fuzzing on coreutils Using LibAFL

—

Report

Valentin Huber

Institute of Applied Information Technology

Zürich University of Applied Sciences ZHAW

contact@valentinhuber.me

June 25, 2024

Abstract

While fuzzing has become an important technique to ensure software reliability and security, the oracles used by most fuzzers to detect software errors are limited to very simple heuristics such as software crashes. To identify more nuanced logic errors, researchers have introduced differential fuzzing, which incorporates checks on the discrepancies between the outputs of two implementations of the same logic under the same input into the fuzzer's oracle. The coreutils suite of programs represents a fundamental component of the UNIX operating system, providing a foundation for users to interact with their operating system through the command line interface. While the version of coreutils developed by the GNU project is the most prevalent, alternative implementations, such as those developed by the uutils project, are also available. The LibAFL is a fuzzing framework was developed with the intention of improving compatibility between improvements to different parts of fuzzers and provides basic support for differential fuzzing.

This work builds upon the foundations laid by LibAFL to create a differential fuzzer that can be run on different implementations of coreutils. It introduces a novel coverage instrumentation on GNU's and uutils' version of coreutils, a coverage gathering and transfer method idiomatic to LibAFL's coverage guidance system, and improvements to the differential fuzzing subsystem. Due to current limitations in LibAFL's input mutators, evaluation was conducted solely on `base64`, where the fuzzer achieved full code coverage and any artificially introduced logic errors were promptly and reliably identified. This system implements a trade-off between completeness and performance, whereby subtle logic errors can be identified by focusing on testing programs' business logic and ignoring functionality that changes the testing environment or is related to `stdio` handling.

Keywords: Software Testing, Fuzzing, Differential Fuzzing, coreutils, LibAFL.

Contents

1 Introduction

Fuzz testing has become an important tool for finding software defects. By repeatedly running a program under test with varying 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 and Adobe, and government agencies such as the US Department of Defense, have developed proprietary fuzzers or contributed to open source fuzzers. Fuzz testing has proven to be an effective tool for both security and reliability testing. It has been credited with finding 20,000 vulnerabilities in Google's Chrome browser alone. [2]

1.1 Differential Fuzzing

While fuzzing has been extensively researched, one area that has not received much attention is the oracle that defines what constitutes an illegal state of the program under test. While simple oracles such as program crashes and timeouts are easy to detect, further research has not gone beyond very general logic, such as memory corruption bugs. [3]

One such oracle is the one 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 for the detection of subtle logic errors that do not invalidate basic guardrails such as memory access rules.

1.2 coreutils

This paper examines coreutils as an example of a target for differential fuzzing. coreutils is a suite of programs that allow users to interact with their system from the command line. Popularised 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 essential tools for many users' daily work. Section 2.1 provides a more detailed introduction to coreutils, including information about its history, interface and technical build-up.

1.3 LibAFL

American Fuzzy Lop (AFL) [4] was an early comprehensive open source fuzzing tool used in countless projects and academic papers. By combining compile-time coverage instrumentation and genetic mutation of inputs that triggered new code paths, it proved to be a very effective at finding software bugs and vulnerabilities [5]. After it ceased to be updated in November 2017, the fork AFL++ [6] has become the de facto replacement.

However, while enhancements to AFL++ have been introduced in many academic and commercial projects, due to the architecture of AFL++, these enhancements have usually not been merged back into AFL++. The result is 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++ have started a new project: LibAFL. It aims to provide a toolkit for building fuzzers that is flexible enough to allow the combination of all these improvements. Refer to Section 2.2 for more details on LibAFL. This work uses LibAFL to build a fuzzer aimed at finding software bugs 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 are the performance trade-offs required for each part?
2. How can the necessary instrumentation be introduced into coreutils? What are the engineering and performance implications of each option?
3. Is it feasible to use LibAFL to build a system with all the logic defined in the answers to the above questions?
4. If so, how effective is the resulting fuzzer at finding bugs in coreutils? What kind of bugs can it find?
5. Can the system be extended to perform differential fuzzing between the different implementations? What changes need to be made?
6. If so, how effective is this second fuzzer at finding bugs in coreutils? What kind of bugs can it find?

Refer to Section 6.1 for a summary of the answers given in this paper.

2 Background

To understand the requirements and architecture of the fuzzer, some background information on coreutils and the architecture of LibAFL is necessary. This section gives an overview of these topics.

2.1 coreutils

On Feb. 8, 1990, D. J. MacKenzie announced fileutils, a suite of utilities for reading and modifying files [8]. A year later, he released textutils (for parsing and manipulating text) [9] and shellutils (for writing powerful shell scripts) [10]. These three collections were merged into one on Jan. 13, 2003, called GNU coreutils. [11] `ls`, `cat`, `base64`, `grep`, `env`, or `whoami`: GNU's coreutils are the foundation of how users interact with most Linux distributions at the command line. [12] Because they are so widely used and central to how users interact with their computers, software quality and the absence of software bugs are particularly important for 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 by default. [13]

2.1.1 Interface

Users interact with coreutils primarily through the command line or in shell scripts. They take different kinds of input, 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`
- 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, numerous alternative implementations have been released. Among these, BusyBox stands out as a notable example. Its objective is to provide a substantial subset of the GNU coreutils, with a particular focus on resource restrictions. Consequently, it is primarily employed in embedded systems [14] or in tiny distributions such as Alpine Linux [15].

In the general trend of rewriting software in memory-safe languages, the uutils project [16] maintains a drop-in replacement implementation of the GNU coreutils written in Rust. This implementation contains all programs, but is still missing certain options. All differences with GNU's coreutils are treated as bugs. Furthermore, the project aims to not only work on Linux, but also on MacOS and Windows.

2.1.3 Build System

The GNU coreutils employ a complex, multi-step build system, including Autoconf [17] and Automake [18]. Any changes to either the code or the build system configuration require a deep understanding of the entire ecosystem to ensure that no unintended changes to the resulting binaries are introduced. The version of coreutils provided by uutils relies on cargo as its build system, which makes the outcomes of changes to the code much more predictable.

2.2 LibAFL

As previously outlined in Section 1.3, LibAFL is an extendible framework for the construction of custom fuzzers. This section will provide a brief introduction to the fundamental concepts and components of LibAFL that are essential for an understanding of this project. For a more comprehensive examination, the authors direct the reader to the original LibAFL paper [7].

2.2.1 Generic Concepts

The core insight of the authors of LibAFL is that the majority of fuzzers contain highly similar components, with advancements in new works typically occurring in only a few of the fundamental fuzzer parts.

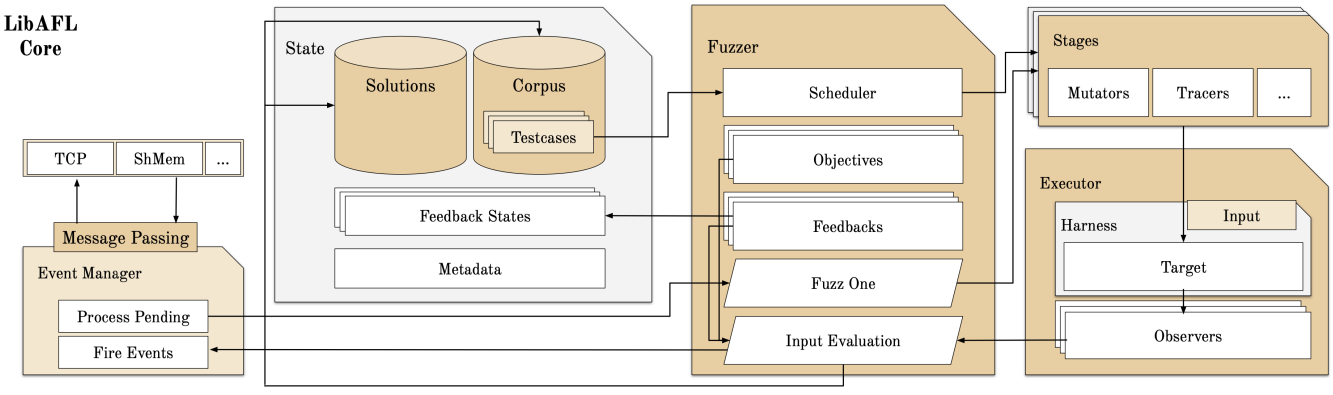

Figure 1: LibAFL's Architecture [7]

This logic is typically introduced into a fork of an existing fuzzer (such as AFL++), which is then abandoned without the advancements being collected in a way that allows them to be reusable and combinable in later projects. An alternative approach is to create a new system from scratch, which introduces a significant amount of duplicate work and code that may not be as optimised as possible. This is because fuzzer parts such as thread synchronisation are not typically the focus of new works.

The authors of LibAFL conducted a comprehensive examination of recent innovations in fuzzing and identified a set of distinct parts present in numerous fuzzers. They then designed the architecture of LibAFL in such a way that authors can modify only the parts they are interested in while relying on a well-optimised default implementation for everything else. Figure 1 illustrates the identified fuzzer parts and their interaction, as implemented in LibAFL. The authors of LibAFL re-implemented the logic introduced in 20 high-impact works to demonstrate that the system is flexible enough to handle logic created independently of 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

From a technical standpoint, the authors' objective was achieved through the utilisation of Rust's trait system in conjunction with generic types. This section will present a selection of these core traits, accompanied by a brief explanation. For a comprehensive list and more comprehensive documentation, the author refers to the paper that introduced LibAFL [7] and the official LibAFL book [19].

Executor The `Executor` is invoked 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 of `Executor`s include `InProcessExecutor`, which calls the provided function directly, and `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` utilise the dynamic instrumentation framework Frida and the emulator QEMU, respectively, 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`, which is used to trace comparisons during an execution run.

Feedback There are two principal applications for `Feedback`s in LibAFL: objectives and feedbacks steering the search. In both cases, a `Feedback` reduces a combination of information provided by the `Executor` and `Observer`s to a Boolean value: Is this test case deemed to be interesting? When used as an objective, this marks the test case in question as a solution. When employed as a steering feedback, it will append the current test case to the corpus from which (after mutating it) the next input is drawn. Typical examples include `CrashFeedback`, `TimeoutFeedback` (both usually used in the objective mode) and `MaxMapFeedback`, which may be utilized to ascertain whether new portions of the memory map utilized to monitor the executed segments of the binary are accessed, thereby representing the coverage-guided aspect of the fuzzer.

Input `Input`s represent data that is mutated and subsequently passed to the `Executor`. It typically comprises a simple variable-length byte array, although it may also be a more complex `struct`. Depending on the specific use case, the former may be sufficient, or the latter may be required to provide additional flexibility.

Mutator Finally, `Mutator`s take an `Input` and modify it in some way. In the case of simple byte array `Input`s, this may involve inverting a bit or inserting an additional byte. 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 initial step is to construct a fuzzer devoid of any features. It merely takes a byte array, randomly mutates it, feeds it to the program under test and checks for crashes. It does not contain any execution steering nor can it trigger the different options. One can select the default implementations for each part outlined in Section 2.2, which reduces the necessary code for the fuzzer to well under 100 lines. However, it is unlikely that this fuzzer will identify any software defects, as it employs a strategy similar to that proposed in the seminal work by Miller *et al.* in 1990 [1]: inserting random data into a program and hoping for a crash, without any additional logic. Furthermore, it does not have access to all parts of the program, as the command line arguments are never used.

3.1.1 Environment Protection

The most straightforward solution to enable 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 potential issue: some programs in the coreutils can modify the environment they are running in, as described in Section 2.1.1. This may be entirely trivial, such as creating and writing to a temporary file in an unrelated part of the filesystem. However, it 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. Firstly, the fuzzer could create a layered filesystem and utilise it to establish an environment for the program under test to run in. This is the methodology employed by Docker, which utilises the host's kernel while deceiving the program under test into believing it runs natively. Any alterations to the file system are recorded and stored, while read operations are responded to with data from the write layer if it has been modified previously and from the host's file system if not. However, the performance implications of this approach are considerable. While a coreutils program takes approximately 20 ms to start on the author's system, a Docker container takes approximately 2 s. Additionally, doing this across many cores, as is typical in modern fuzzing, relies on the paralleliation of starting containers as done by the Docker daemon. This does not necessarily scale linearly with the number of cores, in fact in earlier work by the author [20], sub-linear scaling effects could be observed.

2. An alternative approach would be to introduce a dynamic translation layer that captures the relevant syscalls and handles them appropriately. While this would limit the startup overhead, it would also slow down the program execution. Furthermore, the logic required for dynamic environment protection is complex and may depend on both the program under test and the specific system used to run the fuzzer.

3. Many programs in the coreutils do not alter the environment they run in, or only do so for very specific options. Therefore, while unable to reach all code, limiting the fuzzer to the program parts that do not alter their environment would prevent any performance overhead, but at the cost of completeness. This approach was adopted in this project.

3.1.2 Custom Input Type

This approach necessitates the restriction of the fuzzer to only parts of the program under test, specifically to a white-listed subset of available command line arguments. Consequently, certain components of the command line argument parsing routines will remain untested. It should be noted that the data associated with a specific flag (see Section 2.1.1) may still be invalid, and that the corresponding parsing routine will remain subjected to testing.

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 construct a system where the only additional code required for the testing of new programs is the addition of:

- 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, a system which allows reusing the default byte array mutators for any `Input` part consisting of a byte array is currently in the works in collaboration with the maintainers of LibAFL.

3.2 Optimizations

This basic fuzzer can then be enhanced by the systems that LibAFL provide. With minimal additional code, the fuzzer can be extended to run on all available cores or even multiple machines, utilise advanced algorithms to select the optimal `Mutator`, and so forth. Additional `Observer`s and `Feedback`s, such as a `TimeoutFeedback` can be incorporated 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

In order to test any non-surface-level code, the fuzzer requires information of some form regarding which parts of the binary have just been executed. This coverage information can 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. As previously outlined in Section 2.1.3, modifying the code or build system of GNU coreutils is a challenging endeavour. Previous experiments conducted by the author on coreutils demonstrated that, in principle, the addition of compile-time coverage-gathering instrumentation is feasible. [20]

3.3.1 LibAFL's Coverage Interface

LibAFL heavily relies on shared memory maps for a wide range of internal functionality, including 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` [21] module. This module encompasses 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 to 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` can be employed as `Feedback` in the fuzzer, which prioritizes `Input`s that traverse new paths in the binary under test, given that 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, that is, when the fuzzer and source code to test are compiled into a single unit. In light of the considerations presented in Section 2.1.3, this approach is not a viable solution for this project. Consequently, an alternative methodology was devised.

3.3.2 Instrumentation

Firstly, an implementation of the required function is created based on the simple default implementation provided in the documentation for `SanitizerCoverage` [21]. This implementation marks the map created by the pass in the tested binary as the edges are executed. Additionally, 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 result in the generation of binaries that behave exactly as those produced by an unmodified compilation process, but have additional functionality statically compiled in. This functionality records coverage information and makes it subsequently available through functions callable from dynamically linked binaries.

3.3.3 Dynamic Interface

In a next step, this functionality must be made accessible to the fuzzer. This is achieved by building dynamic system libraries, which are passed to the loader via the `LD_PRELOAD` environment variable. They can hook into the execution process at multiple points to execute their logic. And, since the symbolic information is retained in the binaries under test, they can call the functions providing coverage data described in Section 3.3.2. In this project, two such 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. This size is then printed to `stdout` and read by the fuzzer to allocate its own memory map of the same size. This step is performed only once at the beginning of each fuzzing campaign, since the map size remains constant for a certain program.

Coverage Extraction The second binary performs more complex logic:

1. Prior to the binary being invoked, the fuzzer allocates a shared memory map with the size extracted by the aforementioned method. This map must be accessible to child processes spawned by the fuzzer. To achieve this, `mmap` and `shm_open` based shared memory is used. Additionally, it is necessary to unset the `libc` flag `FD_CLOEXEC` in order 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 serialised to a printable format.

3. The fuzzer only then calls the binary under test, with the second dynamic system library loaded again using `LD_PRELOAD`. In addition to all arguments used in the fuzzing run, the last argument passed is the serialised shared memory description.

4. The helper binary then deserialises the description and makes it available for later use.

5. Furthermore, it also changes `argc` and `argv` to remove the description.

6. With the modified arguments, it then calls `libc_start_main` of the binary under 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 exits, 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 with the exception of the custom `Executor` and some code to set up the shared memory. This makes it highly 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 obtaining coverage information from uutils' version of coreutils is analogous to the one described above. Since Rust uses clang as its underlying compiler, the challenge is reduced to the correct transmission of the compiler and linker flags. 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 are identical to what is described for GNU's coreutils.

3.5 Differential Fuzzing

As outlined in Section 1.1, differential fuzzing describes an advancement in oracle development. It entails executing different implementations of the same logic and comparing their outputs. Typical oracles define fixed logic to determine whether a certain execution should be considered a solution. Typical oracles for this purpose are relatively simple but lack specificity (i.e., they may produce false negatives), such as 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, the creation of such reference implementations is not a viable option for fuzzing alone. Consequently, differential fuzzing is employed almost exclusively when multiple implementations of the same logic are already available. Section 5 presents relevant works in this field.

3.5.1 Existing Functionality and Custom Extensions

LibAFL includes certain existing functionality for differential fuzzing, including

- `DiffExecutor`, which takes two other `Executor`s that are then used to perform the actual executions and whose `ExitKind`s (very coarse exit reason distinction, specifically `Ok`, `Crash`, or `Timeout`) are compared
- `DifferentialObserver`, a trait for `Observer`s 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 three additional `Feedback`s were required:

- `AnyTimeoutFeedback` ignores executions that resulted in a timeout, since cancelled programs may leave partial outputs and thus falsely trigger objective `Feedback`s.
- `DiffExitKindFeedback` checks if the `ExitKind`s of the two `Executor`s 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, the interface of the binary data `Mutator`s in LibAFL is designed in such a way that it only possible to mutate one part of the `struct` in the `Input`. Despite the introduction of significant improvements following a lengthy discussion, the necessary changes were not implemented in time to be considered in this project. Consequently, the evaluation of the produced fuzzer is only possible to a limited extent.

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" [22]. However, this seems to only hold for information written to `stdout`. In its error messages, uutils' programs return additional information, such as a usage tip. Listings 2 and 3 show the differences 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 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 was discovered by an intermediate version of the differential fuzzer. It is required due to an implementation detail: In `base64`, GNU's version prints valid bytes during the decoding process and, once it encounters invalid data, prints an error message. In contrast,uutils' version parses the entire input and before printing the output or error message. An example output can be found 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 for 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 errors 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 fuzzer in its differential fuzzing modes achieves approximately 6000 executions per second. To reduce the impact of bottlenecks from starting many processes, the following experiments are run on single-core invocations of the fuzzer.

4.2.1 Performance Implications of Individual Parts

The fuzzer, as described in Section 3, contains multiple components that collectively influence the overall runtime of a single fuzzer execution. LibAFL contains performance collection code, which provided the results in Listing 6. This analysis demonstrates that the majority of runtime is attributed to the target execution. The remaining functions that contribute significantly to the overall runtime of the fuzzer are the mutations, given that they are executed multiple times on each run, and the coverage analysis, which collectively account for less than 0.5 % of the total runtime.

In its full differential mode on a single thread, the fuzzer reports approximately 250 executions per second, or a runtime of 0.04 seconds per execution. The `TargetExecution` described in Listing 6 comprises multiple steps: the input is written to a file and fed to the binary under test, which is then run. The output of the program is captured, and coverage information is copied back to the fuzzer.

```
Client 001:
  0.0000: Scheduler
  0.0000: Manager
  Stage 0:
    0.0000: GetInputFromCorpus
    0.0005: Mutate
    0.0000: MutatePostExec
    0.9957: TargetExecution
    0.0000: PreExecObservers
    0.0017: PostExecObservers
  Feedbacks:
    0.0000: DiffExitKindFeedback
    0.0015: CombinedCoverage
    0.0000: DiffStdioMetadataPseudoFeedback
    0.0000: StderrNeitherDiffFeedback
    0.0000: TimeoutFeedback
    0.0000: CrashFeedback
    0.0000: StdoutEqDiffFeedback
  0.0005: Not Measured
```

Listing 6: Output from LibAFL's Introspection Modules

Figure 2 compares the performance of GNU's and uutils' versions of coreutils with different approaches to coverage instrumentation. It shows that the performance of the two implementations without instrumentation is comparable, with the uutils version being slightly slower. The compared modes include tainting, or marking which edge has been executed, and counting, which counts how often an edge is passed. The latter requires an additional read access on each edge, and thus is slightly less performant than the first.

Optimization Level	Edge Count
GNU with -O2	801
uutils with `release-small`	14939
uutils with `release-fast`	18928
uutils with `release`	27433

Table 1: Edge Count Depending on Version and Optimization

The addition of instrumentation to GNU's version of coreutils results in approximately a third increase in runtime, whereas this factor is considerably higher for uutils' version. Table 1 illustrates this discrepancy: the edge count is significantly larger for uutils' version compared to GNU's version. The runtime is dominated by the number of edges after a certain point, which explains why the profile optimising for small binary sizes is faster when instrumented compared to the profile optimising for runtime.

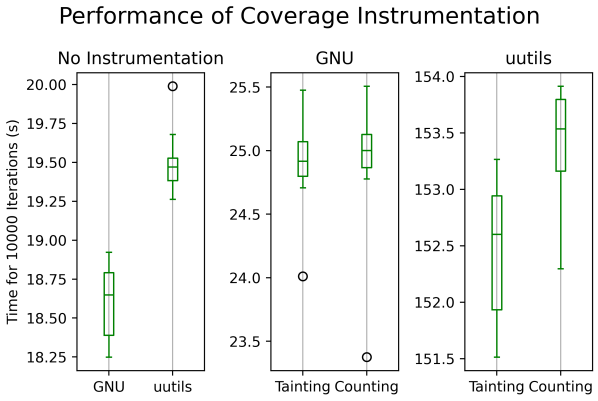

Figure 2: Performance Implications of Coverage Instrumentation

The reason why clang produces a greater number of edges in Rust code compared to C code, at least for the discussed versions of `base64`, could not be evaluated in this project, but may be worthy of further investigation.

As pipes have a maximum size in Linux, the input cannot always be passed to the program under test directly. To circumvent this limitation, the fuzzer will write the input to a temporary file in an in-memory file system and pass this file to the program under test. Figure 3 illustrates the impact of this approach on performance. While the performance impact is minimal for small input data, this is not necessarily the case for larger input data, as illustrated in Figure 4.

Figure 5 illustrates the impact of varying input sizes on the performance of `base64` in total. This includes the additional runtime shown in Figure 4, which demonstrates that the impact of writing the data to a file is an order of magnitude smaller than that of `base64`, and thus acceptable.

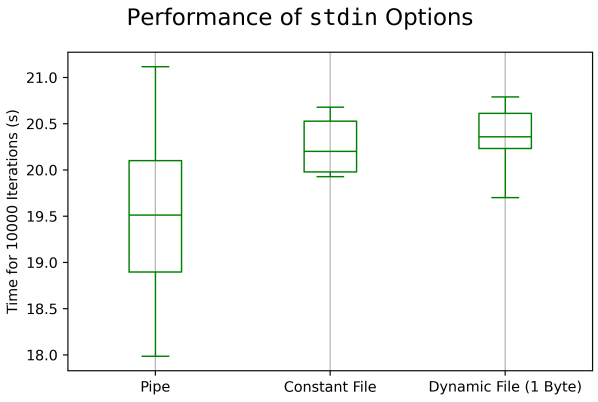

Figure 3: Performance Differences When Passing Input Data to `base64` Directly, with a Default File, and with a File Created on each Execution

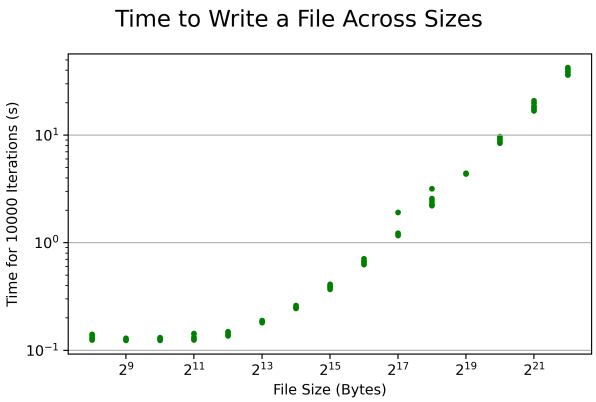

FIGURE 4: Runtime Changes when Writing Differently Sized Files

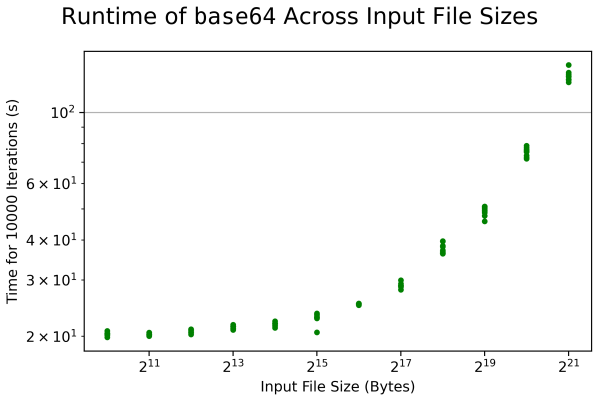

FIGURE 5: Runtime Changes when Executing `base64` on Inputs of Different Sizes

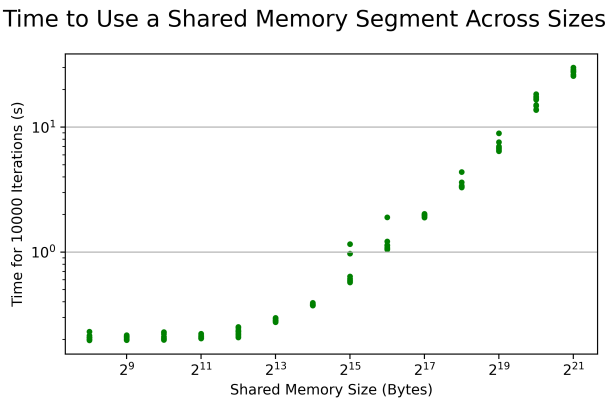

FIGURE 6: Time to Allocate, Persist, Fill, and Check Shared Memory as Provided by `MmapShmem`

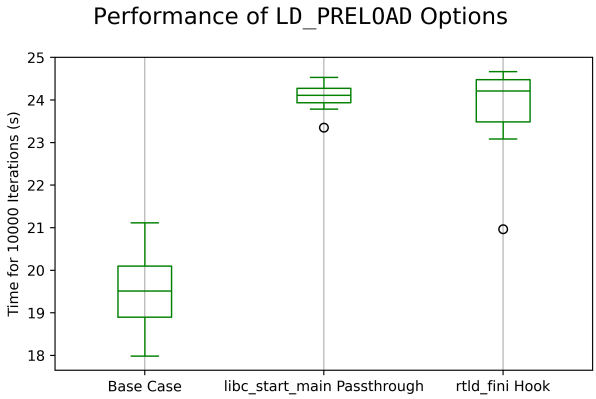

FIGURE 7: Performance Impact of `LD_PRELOAD`

The Performance of shared memory as provided by the `MmapShmem` module in LibAFL is demonstrated in Figure 6 to be comparable to writing an input file, thus eliminating the potential for problematic overhead. Finally, Figure 7 presents the performance implications of hooking into the functions necessary for this fuzzer. Compared to the baseline, the first option merely hooks into `libc_start_main` and calls the program's main function with unaltered parameters. The second option also hooks into `libc_start_main`, but includes a hook for `rtld_fini` in its call to the main function. The hook in this example only calls the original `rtld_fini`, whereas the fuzzer uses this hook to copy coverage information to the shared memory.

The overall performance of the fuzzers can be explained almost entirely by a combination of the aforementioned experiments, with the remaining runtime being introduced by minor additional logic such as output capturing.

4.2.2 Coverage

According to the fuzzer, only 9.854% of edge coverage is reached in its differential mode. 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 `Inputs`, 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` evaluates all `Inputs` as interesting 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.

An analysis of the results reported by `gcov` shows that 61.49% of 161 lines were executed. After manually invoking `base64` with its `--help` and `--version` flags, gcov reported 83.23% coverage. Manual inspection of the coverage file showed that the missing lines consist of `stdio` error handling code (e.g. write operations to `stdout`, which should never fail, since pipes are not intentionally broken by the fuzzer).

4.2.3 Performance on Individual Implementations

The fuzzer is constructed in a manner that allows it to be run on a single implementation. As described in Sections 1.1 and 3.5, this reduces the kind of software error that can be found to those that result in the program under test crashing. Given that the runtime of `base64` it is not possible to discover infinite loops based on program runtime, since the execution timeout would need to be impractically long to ensure the absence of false positives.

As anticipated from the experiments shown in Figure 2, the fuzzer performed significantly better on GNU's version of coreutils, achieving a 5.1 times higher execution rate. The discrepancy in the execution speed between the fuzzer and manual tests can be attributed to the additional computational processes inherent to the fuzzer, such as coverage analysis. The enhanced child process management in `CommandExecutor`, as employed by the fuzzer, further optimises the execution speed, reaching a point where the fuzzer achieved almost twice the execution count compared to manual tests within the same time frame.

The fuzzer did not identify any input that could cause crashing errors in either the GNU or uutils versions of coreutils.

4.3 Discovered Errors

In an evaluation run spanning 24 hours and encompassing nearly 100 million executions, the fuzzer was unable to identify any errors in `base64` from either version of coreutils, neither in its individual or differential mode. However, GNU's version of `base64` comprises 161 lines of code according to `gcov` including its user interface. Given its extensive usage and the number of individual who have tested the code, it is highly improbable that there are any errors in the code base to begin with.

To assess the efficacy of this fuzzer in identifying logic errors, an artificial bug was introduced into the source code of GNU's version of `base64` (see Listing 7). This bug does not result in the program crashing or entering a state that would be flagged as a bug by conventional fuzzers. The fuzzer developed in this project identified this bug within minutes on multiple runs. This demonstrates that even bugs that have complex requirements on the input that trigger them can be found with this approach.

```
1  void base64_encode(const char *restrict in,
       ↪ idx_t inlen, char *restrict out, idx_t
       ↪  outlen)
2  {
3  if (inlen == 3)
4      if (in[0] == 'a')
5          if (in[1] == 'b')
6              if (in[2] == 'c')
7                  printf("Hello, World!");
8
9  [remaining function code]
10 }
```

Listing 7: Inserted Artificial Bug

5 State of the Art

This is not the first project to employ differential fuzzing techniques nor the first to fuzz coreutils. This section introduces previous works and discusses their limitations compared to this project.

5.1 uutils' Fuzzing Sub-Project

The only other project that performs differential fuzzing on coreutils is within the uutils coreutils repository. It contains code to fuzz 16 programs, 11 of which perform differential fuzzing against the system installation of the appropriate program from GNU coreutils.

libFuzzer is used as a fuzzing backend, which integrates with cargo and provides coverage instrumentation. However, the interface of libFuzzer is currently limited to the `fuzz_target` macro, which only supports input types in the shape of simple byte arrays. In the project in its current state, this input is not used in combination with differential fuzzing. For those programs, the input is manually randomly generated, thus negating any advantage that coverage-guided fuzzing introduces.

Only for the non-differential fuzzers, the input provided by libFuzzer is used. Of the five fuzzing binaries in this category, four do not test entire programs but instead import parsing routines. The sole exception is `date`, where actual coverage-guided fuzzing on a full program from coreutils is performed. The command line arguments are generated from the byte array input by splitting at `NULL` bytes. This is not a generally viable strategy, as is argued in Section 3.1.1.

Overall, compared to uutils' fuzzing sub-project, this work performs both coverage-guided and differential fuzzing simultaneously, while utilizing the flexibility gained by building on top of LibAFL to increase performance and ensure environment protection and thus reliable testing.

5.2 Fuzzing coreutils

The seminal work that introduced 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 undertaking this approach, the researchers identified errors in between a quarter and a third of utilities, depending on the UNIX system. [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. Additionally, the authors tested the uutils implementation and found similar levels of error prevalence. [23]

5.2.1 Symbolic Execution Frameworks

KLEE is a symbolic execution-based whitebox fuzzer that relies on SMT solvers to calculate inputs that will traverse new paths in a program. This approach has been shown to be highly effective at achieving high coverage in both the GNU and BusyBox versions of coreutils. [24] Additionally, KLEE forms the foundation for a list of fuzzers that build upon its capabilities to introduce novel concepts. Some notable examples include:

- CRETE introduced the concept of concolic execution, whereby the program under test is run with concrete values and the instructions executed are traced to subsequently calculate new inputs based on this instead of keeping the entire state-space in memory. [25]
- FUSE only selectively performs symbolic execution based on function complexity analysis. [26]
- Learch introduced a state selection algorithm that learns which state contributes most towards maximizing coverage. [27]
- Cloud9 proposed a custom state-merging algorithm to help mitigate the state-explosion problem inherent in symbolic execution. [28]

5.2.2 Other approaches

Outside of symbolic execution-based fuzzing, coreutils have received little attention in academia. The following works were identified:

- CLIFuzzer [29] 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" [30]. Sjöbom and Hasselberg attempted to evaluate its effectiveness, but showed the drawbacks outlined in Section 3.1.1. [31]

5.2.3 Other Related Works

Finally, certain loosely fuzzing-related projects make use of 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. [32]
- 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. [33]
- IMF-SIM is an attempt to de-obfuscate programs by comparing a vector calculated based on the behaviour of them. This vector is generated by a fuzzing-like process. The authors evaluate their approach on coreutils. [34]
- 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. [35]

5.3 Differential Fuzzing

Since differential fuzzing can only be performed on targets where two implementations are available that claim to include the exact same functionality, certain categories of such programs appear in literature.

5.3.1 Network Protocols

Given that the interface of network protocols must be well-defined for compatibility between different systems, and that the diverse ecosystem of network-attached systems produces multiple implementations of software stacks to handle such network protocols, this is an obvious target for differential fuzzing. Examples include testing Deep Packet Inspection (DPI) evasion for QUIC [36], errors in recursive DNS resolvers [37], or request smuggling through HTTP proxies [38]. Additionally Zheng *et al.* extract finite state machines from different implementations of 14 network protocols and perform differential fuzzing to find errors in them. [39]

5.3.2 Cryptography Libraries

Cryptography libraries represent another potential target for differential fuzzing, given that the underlying logic is well-defined and multiple implementations exist (as evidenced by the 37 different implementations tested in [40]). Examples include TLS handshake testing [40], [41], or TLS/SSL certificate validation [42].

5.3.3 Compilers and Interpreters

Despite the similarities between different JavaScript engines, there remain significant differences, due to unspecified behavior. However, Bernhard *et al.* perform differential fuzzing on JavaScript engines by comparing their interpreter to their just-in-time (JIT) compiler. [43]

In [44] and [45], the authors test different Java Virtual Machine (JVM) implementations against each other to find 92 bugs in total. Hamidy tests different WebAssembly interpreters [46], while Li and Su attempt to find undefined behavior based bugs in compilers [47]. Another project trained a machine learning system to generate realistic programs as inputs for different compilers. [48]

Similarly, EVMFuzz compares different implementations of the Ethereum Virtual Machine. [49] Fluffy attempts to create multi-transactional test cases for the Ethereum network that exploit consensus bugs in different implementations to trigger hard forks of the network. [50] Other networks tested include the Neo blockchain in NeoDiff. [51]

5.3.4 Regression Testing

Differential fuzzing can also be employed in the context of regression testing. This approach has the significant advantage of not necessitating two different implementations of the same logic, as two versions of a program are tested against each other. This methodology is only viable if the difference between the tested versions is limited to performance enhancements or other refactoring changes, as any alterations to functionality or the introduction or fixes of bugs would be caught by the fuzzer. Examples include work done by Noller *et al.* [52].

5.3.5 Deep Learning

Deep learning systems remain still largely only superficially understood. Ensuring reliable operation is becoming ever more important as such systems are increasingly used in high-stakes environments such as self-driving cars. While the same approach of comparing multiple similar tools can be applied to deep learning systems as well, in works such as by Guo *et al.* [53], the term differential fuzzing refers to the technique of adding minor noise to an input and checking if a classifier model returns the same output for the original and altered input. Similar to what is done in this project, coverage-guided fuzzing can be applied to deep learning models by counting activated neurons instead of executed edges.

5.3.6 Side-Channel Attacks

Finally, a different form of differential fuzzing can be employed to discover side-channel attacks. By comparing the resource consumption (e.g. time, memory, or energy) of the same program using two different inputs and mutating these inputs to maximise the difference, information about the program state can be retrieved, as demonstrated in [54].

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 in 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.

1. In principle, all components of coreutils can be tested using a fuzzer. However, based on the interface information provided in Section 2.1.1 and the considerations on environment protection in Section 3.1.1, a fuzzer designed to achieve complete coverage must introduce a computationally expensive layer to record environmental changes and prevent their persistence across executions of the program under test. Furthermore, a layer to emulate failed system interactions (such as broken pipes) must be implemented. The performance cost of these layers renders them impractical for use in testing without the availability of very large compute resources.

2. This work introduces a novel approach to LibAFL for gathering 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.

3. 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 experiments on custom `Input` types (as described in Section 3.1.2 and discussed in Section 6.3.3).

4. A non-differential fuzzer is only able to 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 4.2.3 demonstrates that even long evaluation runs on `base64` did not uncover any such bugs.

5. Section 3.5.1 outlines the existing functionality in LibAFL employed by the fuzzer in its differential mode, along with the custom extensions introduced during this project. Given 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 feasible but also a relatively straightforward engineering task.

6. Finally, the fuzzer in its differential mode is able to identify and locate artificially introduced non-trivial logic errors in the different implementations that are not discoverable by its non-differential mode. It explores all program parts, with the exception of those that have been excluded based on environmental protection reasons, as previously described.

6.2 Contributions

During the course of this project, a number of enhancements 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)
- A bugfix for `MmapShmemProvider` (PR #2298)
- Fixes to the handling of `StdOutObserver` and `StdErrObserver` and an example fuzzer to demonstrate their usage as part of a refactoring in collaboration with LibAFL's maintainers (PR #2311)

Based on feedback by the author of this work, the author in collaboration with LibAFL's maintainers began 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

Although the fuzzer introduced in the previous sections was already capable of producing results in the form of proof-of-concept work, it is subject to a number of significant limitations.

6.3.1 Untested Program Parts

Sections 2.1.1 and 6.1 examine the limitations of the fuzzer in terms of its inability to test functionality that alters the environment. Furthermore, the handling of errors in the input and output of the programs (e.g. broken pipes) is not tested due to the absence of an intermediate layer that could introduce these errors artificially. Future projects may introduce these missing layers either individually or in combination, with the aim of testing the code sections not tested by this project.

6.3.2 Instrumentation Performance on uutils' coreutils

Section 4.2.1 discusses that while the performance between GNU's and uutils' version of coreutils is similar without instrumentation, adding coverage gathering logic increases the runtime of uutils' version by a factor of 8, compared to 1.3 for GNU's version. Future work may investigate the reason for this difference to improve this fuzzer's performance.

6.3.3 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

This project introduces a new coverage-guided greybox fuzzer that performs differential fuzzing on coreutils. It builds on top and includes many advanced algorithms implemented in LibAFL while limiting the inputs it generates to those that ensure a constantly stable testing environment.

A novel approach to transferring coverage information from a program executed as a separate process is employed, whereby the programs under test are extended with an interface to dynamically access coverage information without any changes to the source code. The system dynamically loads and links helper binaries during execution, providing coverage information in a manner that is idiomatically readable by LibAFL's coverage guidance modules. Due to incomplete functionality in LibAFL, the fuzzer produced in this project could only be evaluated on base64, where it achieved full coverage but did not discover any software errors. Further extensive tests will be required as a base for a full evaluation once the missing logic has been added to LibAFL.

In the interest of open science, this project is released under an open-source license. During development, over 500 lines of code have been introduced to the upstream project, with further major changes already in discussion. The source code of the project is publicly available at

https://github.com/riesentoaster/coreutils-differential-fuzzing.

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." (n.d.), [Online]. Available: https://lcamtuf.coredump.cx/afl/#bugs (visited on Jun. 3, 2024).

[6] "AFL++." (n.d.), [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." (n.d.), [Online]. Available: https://www.busybox.net/about.html (visited on May 22, 2024).

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

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

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

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

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

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

[34] 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.

[35] 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.

[36] G. S. Reen and C. Rossow, "DPIFuzz: A differential fuzzing framework to detect dpi elusion strategies for quic," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20, ¡conf-loc¿, ¡city¿Austin¡/city¿, ¡country¿USA¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2020, pp. 332–344, ISBN: 9781450388580. DOI: 10.1145/3427228.3427662. [Online]. Available: https://doi.org/10.1145/3427228.3427662.

[37] J. Bushart and C. Rossow, "Resolfuzz: Differential fuzzing of dns resolvers," in *Computer Security – ESORICS 2023*, G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis, Eds., Cham: Springer Nature Switzerland, 2024, pp. 62–80, ISBN: 978-3-031-51476-0.

[38] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda, "T-Reqs: Http request smuggling with differential fuzzing," 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. 1805–1820, ISBN: 9781450384544. DOI: 10.1145/3460120.3485384. [Online]. Available: https://doi.org/10.1145/3460120.3485384.

[39] M. Zheng, Q. Shi, X. Liu, *et al.*, "ParDiff: Practical static differential analysis of network protocol parsers," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. DOI: 10.1145/3649854. [Online]. Available: https://doi.org/10.1145/3649854.

[40] A. Walz and A. Sikora, "Maximizing and leveraging behavioral discrepancies in tls implementations using response-guided differential fuzzing," in *2018 International Carnahan Conference on Security Technology (ICCST)*, 2018, pp. 1–5. DOI: 10.1109/CCST.2018.8585565.

[41] A. Walz and A. Sikora, "Exploiting dissent: Towards fuzzing-based differential black-box testing of tls implementations," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 2, pp. 278–291, 2020. DOI: 10.1109/TDSC.2017.2763947.

[42] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "NEZHA: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 615–632. DOI: 10.1109/SP.2017.27.

[43] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "JIT-Picking: Differential fuzzing of javascript engines," 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. 351–364, ISBN: 9781450394505. DOI: 10.1145/3548606.3560624. [Online]. Available: https://doi.org/10.1145/3548606.3560624.

[44] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," *SIGPLAN Not.*, vol. 51, no. 6, pp. 85–99, Jun. 2016, ISSN: 0362-1340. DOI: 10.1145/2980983.2908095. [Online]. Available: https://doi.org/10.1145/2980983.2908095.

[45] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1257–1268. DOI: 10.1109/ICSE.2019.00127.

[46] G. Hamidy, "Differential fuzzing the webassembly," M.S. thesis, Aalto University, Aug. 18, 2020.

[47] S. Li and Z. Su, "Finding unstable code via compiler-driven differential testing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, ¡conf-loc¿, ¡city¿Vancouver¡/city¿, ¡state¿BC¡/state¿, ¡country¿Canada¡/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2023, pp. 238–251, ISBN: 9781450399180. DOI: 10.1145/3582016.3582053. [Online]. Available: https://doi.org/10.1145/3582016.3582053.

[48] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 95–105, ISBN: 9781450356992. DOI: 10.1145/3213846.3213848. [Online]. Available: https://doi.org/10.1145/3213846.3213848.

[49] Y. Fu, M. Ren, F. Ma, *et al.*, "Evmfuzz: Differential fuzz testing of ethereum virtual machine," *Journal of Software: Evolution and Process*, vol. 36, no. 4, e2556, 2024. DOI: https://doi.org/10.1002/smr.2556. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2556. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2556.

[50] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, USENIX Association, Jul. 2021, pp. 349–365, ISBN: 978-1-939133-22-9. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/yang.

[51] D. Maier, F. Fäßler, and J.-P. Seifert, "Uncovering smart contract vm bugs via differential fuzzing," in *Reversing and Offensive-Oriented Trends Symposium*, ser. ROOTS'21, Vienna, Austria: Association for Computing Machinery, 2022, pp. 11–22, ISBN: 9781450396028. DOI: 10.1145/3503921.3503923. [Online]. Available: https://doi.org/10.1145/3503921.3503923.

[52] Y. Noller, C. S. Păsăreanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "HyDiff: Hybrid differential software analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020, pp. 1273–1285, ISBN: 9781450371216. DOI: 10.1145/3377811.3380363. [Online]. Available: https://doi.org/10.1145/3377811.3380363.

[53] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "DLFuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 739–743, ISBN: 9781450355735. DOI: 10.1145/3236024.3264835. [Online]. Available: https://doi.org/10.1145/3236024.3264835.

[54] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "DiffFuzz: Differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 176–187. DOI: 10.1109/ICSE.2019.00034.