Running KLEE on GNU coreutils

Valentin Huber

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

February 11, 2024

Contents

1 Introduction

KLEE [1] is an open source, symbolic execution based, advanced fuzzing platform. It was introduced in the seminal paper titled "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs" in 2008. In their article, Cadar *et al.* present their work and evaluate it on a diverse set of programs. The most prominent of those is the GNU coreutils suite, in which ten fatal errors were found.

Ever since then, KLEE has not only matured as a fuzzer, it has also been used extensively as a platform for other researchers to build on top of, as I have discovered in [3]. As an introduction to the practical side of fuzzing, I attempted to answer the following questions about KLEE:

1. Reproducing the original paper (see Section 3)
   (a) Can the current version of KLEE be run on coreutils version 6.10 (as tested in the original paper)?
   (b) Can the same metrics as measured in the original paper still be measured?
2. Examining the statistical distribution of results over different fuzzing times (see Section 4)
   (a) How does the non-determinism in KLEE influence the variance in the results between different test runs?
   (b) How do the measured metrics compare to what was published 15 years ago?
   (c) How do different testing timeouts influence results?
3. Testing more recent versions of coreutils (see Section 5)
   (a) What needs to change in the test setup to test more recent versions of coreutils?
   (b) How do the results from testing different versions of coreutils differ?

All experiments were run on a virtualized server with the following specs: AMD EPYC 7713 64C 225W 2.0GHz Processor, 1 TiB RAM, 2x 25GiB/s Ethernet.

2 Background

What follows is a short explanation of the application of symbolic execution in fuzzing. For more extensive background, I refer to some of my previous work [3], [4].

Remember that KLEE is an open-source, symbolic execution based fuzzer. It takes LLVM bytecode from the program under test (PUT) as its input, runs its analysis on it. KLEE then outputs some statistics about the run, inputs to the PUT that crash it, and inputs that, when executed, cover all branches KLEE has examined during its analysis.

2.1 A Primer on Symbolic Execution

KLEE is a fuzzer based on symbolic execution. This means that instead of executing a PUT with a concrete value, it instead runs through the instructions and maps relationships between data in memory (such as variables and user input) to mathematical formulas. So an instruction like `%result` ↪ `= add i32 %a, %b` would be mapped to the logical relationship $\phi_k = \phi_i + \phi_j$. Conditional jumps are mapped to conditions on these variables for both outcomes of the condition, so the instruction `%isPositive = icmp sgt i32 %result, 0` would be represented with $\phi_k > 0$ and $\phi_k \leq 0$ respectively.

The set of all conditions along a certain path through the PUT are called the *path condition*. It can be passed to s satisfiability modulo theories (SMT) solver (KLEE uses STP [5] as a default), which returns values for all user inputs, such that the PUT is forced down the exact path represented by the path condition.

This is the major advantage of symbolic execution based fuzzing, as compared to ordinary fuzzing. By not using concrete values, but instead logical representations of user input, it essentially runs through the PUT with all possible user inputs *at the same time*. So if the solver returns that no inputs satisfy the passed formula, we have proven that such inputs simply do not exist. To be able to do this, it accepts the huge overhead of translating the code to formulas and then solving them.

2.2 Symbolic Execution in Practice

Symbolic execution in fuzzing has several major challenges to overcome. I have previously discussed them in detail [3], but would like to give a short summary here:

- Environment interactions (such as file system interactions) in general are opaque to the fuzzer and cannot be mapped to logical formulas. KLEE deals with this by solving the path constraint before the instruction in question and then uses concrete values in the call. This abandons the claim on completeness symbolic execution typically has, but is often the only feasible way to still continue analysis.

- The second major challenge in symbolic execution is what is known as *path explosion*. Because the number of program states grows exponentially with the number of instructions, for all but the most simple programs it is not feasible to calculate the entire state space. KLEE deals with this by reducing the search space to actually executable instructions, using advanced data structures, and examining paths through the PUT consecutively, with a user-defined timeout. To maximize the state space and code coverage as quickly as possible, it alternates between two strategies for choosing the next input to evaluate: KLEE either chooses the input that promises to increase the coverage the most and a random input to prevent the execution from getting stuck in a certain subtree of the PUT.

- KLEE needs to model the entire memory of a process. This is straight-forward as long as variables are used directly but becomes a challenge when pointers are involved. This is especially true if the value of these pointers depends on user input, since this would require KLEE to model all possible addresses having all possible values, which instantly explodes the memory consumption and number of states to examine and is thus infeasible. KLEE deals with this by representing such pointer operation as array accesses where the accessed object is copied as often as necessary to model all possible results, including error states.

- As programs become more complex, the path constraints become increasingly long and solving them contributes more and more to the fuzzer's runtime. KLEE applies some advanced optimizations, like query splitting and more general optimizations, or a cache of previous results, which often solve supersets the query they are a solution to. Finally, KLEE defines a timeout, after which the solver is interrupted and analysis is continued at an other branch.

3 Reproducing the Original Paper

I'm basing my experiment setup on the original paper [2], the FAQs in the project's documentation [6] and the tutorial on testing coreutils version 6.11 [7].

3.1 Project Setup

KLEE is a complex system including complex dependencies such as the SMT solvers. The maintainers provide a Dockerfile and the corresponding Docker image. Using Docker as an intermediate form of virtualization adds a layer of indirection and a performance penalty. However, since I'm not necessarily interested in maximizing performance in this project, but instead focus on comparing different setups, this is a tradeoff worth taking. Using Docker to evaluate fuzzers' performance has been done before [8]. Finally, this makes complex build steps reproducible and acts as documentation.

3.2 Naïve Approach

When attempting to build coreutils 6.10 directly in the current version of KLEE's Docker image, I ran into an issue: The Docker image is based on Ubuntu 22 (Jammy), and no longer is able to build coreutils 6.10 with the GNU Compiler (GCC). This is because coreutils' build system attempts to detect what system it is running on, and the variable the detection is based on is no longer defined. Specifically, in `freadahead.c` the following check is performed:

```
#if defined _IO_ferror_unlocked      /* GNU libc, BeOS */
```

The error message and the full `freadahead.c` can be found in Appendix 1.

3.3 Using an Old Version of Ubuntu

One attempt to mitigate this issue would be to rewrite this check to allow the version of GCC installed on KLEE's Docker image to compile coreutils 6.10. However, I opted to pursue a different avenue, because of two reasons:

1. Build systems are not my area of expertise and I do not know how many other issues would appear once the first was solved.
2. Changing code always adds risk of introducing additional software errors, which would distort my findings.

Therefore, I attempted to build the binaries on an old version of Ubuntu, and then move the binaries to KLEE's Docker image. Specifically, I chose the latest LTS version which was available when version 6.10 of coreutils was current. This approach worked without any additional changes to the code nor the build system. The setup of the Docker image then used to build coreutils can be seen in Listing 1.

3.4 Generating LLVM Bytecode Files

Building binaries themselves is unfortunately not enough, since KLEE does not take pure binaries as its input, but instead requires LLVM bytecode. Compiling an ordinary `.c` file to LLVM can easily be done using `clang`. However, again, coreutils use a complex build system which means to just use `clang`, I'd have to deeply understand and modify it, with the drawbacks listed above.

Simply passing `clang` as the C compiler to the build system does not work, since the produced output is not a runnable binary, and the build system requires the compiler's output to be executable.

Fortunately, there exists Whole Program LLVM (WLLVM) [9], a tool specifically designed to work with complex build systems while still producing LLVM bytecode as one of its outputs. This is achieved by injecting its compiler into the build system. The compiler creates executable binaries and

Listing 1: Dockerfile content to prepare a system for building coreutils 6.10

```
# =======================================
# base
# =======================================

FROM ubuntu:14.04 as klee-coreutils-base

# installing dependencies
RUN apt-get update \
    && apt-get install -y \
    wget \
    build-essential

# downloading source code
RUN wget "http://ftp.gnu.org/gnu/coreutils/coreutils-6.10.tar.gz" \
    && tar xf "coreutils-6.10.tar.gz" \
    && mv "coreutils-6.10" coreutils

# modifying source code according to the documentation of the original experiment
RUN sed -i \
    's/^#define INPUT_FILE_SIZE_GUESS (1024 \* 1024)$/#define INPUT_FILE_SIZE_GUESS
    ↪ 1024/g' \
    coreutils/src/sort.c
```

additionally injects LLVM bytecode into a dedicated section of the object files. In a second step, these then get extracted and linked together to produce LLVM bytecode files.

Since I'm running WLLVM on an old version of Ubuntu, I was forced to use an old version of WLLVM as well, because newer versions require a version of python which is not available on Ubuntu 14.04. To create proper input files for KLEE, I added two options, to reduce warnings (`--build`) and to turn off premature optimizations according to the KLEE documentation (`CFLAGS`) [7].

The Dockerfile section to build the LLVM bytecode can be found in Listing 2.

3.5 Coverage Data Gathering

A simple way to compare what these experiments accomplish compared to the experiments documented in the original paper is to look at coverage data, specifically coverage as measured by `gcov`. To gather this information, one needs to compile the binaries using GCC, and tell the compiler to add coverage gathering instrumentation. Along with the binaries, a note document (`<executable-name>.gcno`) is created. When the binary is executed, the added instrumentation records which path through the code is taken and, together with information from the notes file, stores its results in a coverage data file (`<executable-name>.gcda`). This file can then be analyzed with `gcov` to get human-readable coverage data.

With this step however, I ran into the same issue as before: Recent versions of GCC no longer build coreutils 6.10. I adopted the same approach and used the same base image as described in Section 3.3. The Dockerfile excerpt with the build step can be found in Listing 3.

I made two changes compared to building the LLVM bytecode files, to increase the accuracy of the measurements:

- I replaced all calls to `_exit` with calls to `exit`, so that those instructions are also included in the measurements. This was done according to the instructions in the FAQ [6].
- The original paper mentions that coverage is measured only on executable lines of code. Specifically, Section 5.1 of the original paper says

  > "We measure size in terms of executable lines if code (ELOC) by counting the total number of executable lines in the final executable after global optimization, which

Listing 2: Dockerfile content to build coreutils to LLVM bytecode using WLLVM

```
# ========================================
# llvm
# ========================================

FROM klee-coreutils-base as klee-coreutils-llvm

# installing dependencies
RUN apt-get install -y \
    clang \
    llvm \
    python3-pip

# Newer versions are no longer compatible with the latest python version available on
    ↪ Ubuntu 14.04
RUN pip3 install --upgrade -v "wllvm==1.1.5"

ENV LLVM_COMPILER clang
ENV CC wllvm

# compiling code to llvm bytecode (.bc)
WORKDIR /coreutils/obj-llvm
RUN ../configure \
    --build x86_64-pc-linux-gnu \
    --disable-nls \
    LLVM_COMPILER=clang \
    CC=wllvm \
    CFLAGS="-O0 -D__NO_STRING_INLINES -D_FORTIFY_SOURCE=0 -U__OPTIMIZE__" \
    && make \
    && make -C src arch hostname

# extracting llvm bytecode from object files
WORKDIR /coreutils/obj-llvm/src
RUN find . -executable -type f | xargs -I '{}' extract-bc '{}'
```

Listing 3: Dockerfile content to build coreutils instrumented to record coverage

```
# ========================================
# gcov
# ========================================

FROM klee-coreutils-base as klee-coreutils-gcov

# compiling code to binaries instrumented with gcov
WORKDIR /coreutils/obj-cov
RUN ../configure \
    --build x86_64-pc-linux-gnu \
    --disable-nls \
    CFLAGS="-O2 -g -fprofile-arcs -ftest-coverage" \
    && find .. -type f -name '*.c' -exec sed -i -E 's/\b_exit\(/exit(/g' {} + \
    && make \
    && make -C src arch hostname
```

Listing 4: Dockerfile content to prepare the fuzzing stage

```
# =======================================
# exec
# =======================================

FROM klee/klee AS klee-coreutils-exec

# setting up klee env
RUN wget "http://www.doc.ic.ac.uk/~cristic/klee/testing-env.sh" \
    && env -i /bin/bash -c '(source testing-env.sh; env >test.env)' \
    && wget "http://www.doc.ic.ac.uk/~cristic/klee/sandbox.tgz" \
    && tar xzfv sandbox.tgz \
    && mv sandbox.tgz /tmp \
    && mv sandbox /tmp

# copying files from build stage
COPY --from=klee-coreutils-llvm --chown=klee /coreutils/ ./coreutils-llvm/
COPY --from=klee-coreutils-gcov --chown=klee /coreutils/ ./coreutils-gcov/

# copying run scripts
COPY analyze.sh ./

# setting default values for analyze script
# can be overridden using -e in docker run
ENV KLEE_MAX_TIME_MIN 60
ENV UTIL echo
ENV SKIP_KLEE_ANALYSIS ""

CMD bash ./analyze.sh \
    --llvm-dir ./coreutils-llvm/obj-llvm/src \
    --cov-dir ./coreutils-gcov/obj-cov/src \
    --skip-klee-analysis "${SKIP_KLEE_ANALYSIS}" \
    --klee-max-time "${KLEE_MAX_TIME_MIN}" \
    --out-dir ./out \
    "${UTIL}"
```

eliminates uncalled functions and other dead code." [2]

I am not sure how Cadar *et al.* calculated the executable lines of code, since this is not trivial. I did enable normal global optimization (`-O2`), but this may still result in a considerable underestimation of coverage.

3.6 Preparing KLEE

Finally, the bytecode files can be passed to KLEE for the actual fuzzing. To prepare KLEE's Docker image, the environment and sandbox are prepared according to the documentation [6]. Then, the bytecode files from the step outlined in Section 3.4 and the binaries instrumented with `gcov` along with their notes files are copied to the analysis image. The analysis step itself is an involved process itself and is done by executing a shell script (`analyze.sh`). This step is explained in Section 3.7. The analysis script is copied into the image and executed on container start. To allow passing certain settings to the analysis step, environment variables are used,which can be set in the `docker run` command.

The Dockerfile excerpt for this step can be found in Listing 4.

3.7 Running KLEE

When starting the Docker image built with the steps outlined before, a shell script is executed. This script handles the evaluation settings, input and output files, and collects metrics. Specifically, the following steps are performed:

1. The input including the passed settings are parsed. The script allows setting input and output directories (`--llvm-dir`, `--cov-dir`, `--out-dir`), KLEE's timeout (`--klee-max-time`), and skipping the fuzzing step (`--skip-klee-analysis`). The latter allows gathering additional metrics without based on the output from a previous fuzzing run without having to perform additional, computationally expensive analysis.
2. To run KLEE, the analyst is required to pass arguments setting the size and number of inputs and input files to be tested. For most coreutils, this is the same, however (as mentioned in Section 5.2 of the original paper [2] and explained in the FAQs [6]) some utils need different settings to achieve a decent coverage. The analysis script assembles the command to run KLEE, including the constant settings, util-dependant settings, and the timeout set in the script arguments.
3. Then, the actual fuzzing is performed.
4. KLEE's output is examined in a few ways:
   (a) For each found error, human readable outputs are created using `ktest-tool`.
   (b) `klee-stats` is invoked to export metrics collected by KLEE.
   (c) All test cases generated by KLEE are used as input for `klee-replay` pointed at the binary instrumented with coverage gathering instructions. This ensures that each instruction analyzed by KLEE during its fuzzing is also executed and thus recorded in the coverage results. Since the instrumented binaries were not compiled on the same system as they are executed on, the environment variables `GCOV_PREFIX` and `GCOV_PREFIX_STRIP` need to be set appropriately.
5. Finally, certain large output text files are compressed to minimize disk usage.

3.8 Extracting Human-Readable Coverage Data

As a last step, the output of the binaries instrumented to gather coverage metrics needs to be fed back into `gcov`. Unfortunately, the format of these output files changed at some point and the version of `gcov` installed in KLEE's Docker image is no longer able to read them. They are therefore fed back into the Docker image that created them, where an obviously compatible version of `gcov` is available.

3.9 Gathered Metrics

In each run on each util of the experiment as discussed above, the following metrics were collected:

- The util name including the version and the run
- The timeout passed to KLEE
- The number of errors as reported by KLEE by type according to the file extension:
  - `ptr` errors, e.g. invalid pointers, null page accesses, out of bound pointers
  - `exec` errors, which appear on illegal instructions and external calls with a symbolic `errno` call
  - `model` errors, which appears when a symbolic size is concretized
  - `solver` errors, which are query timeouts
  - `abort` errors
  - The total number of errors
- Instruction and branch coverage[1] as reported by KLEE[2]
- Coverage as measured by `gcov`

[1]KLEE includes library code in its coverage numbers, and thus reports significantly lower coverage compared to `gcov`.

[2]KLEE reports a set of additional metrics like time spent in the solver, number of instructions analyzed and number of cache hits, but these were not examined further in this paper.

4 Comparing Runs

Figure 1 shows the spread of the measurements between four runs of the experiments with the same parameters. The non-deterministic nature of the results is due to the inherent non-determinism of the fuzzing process in KLEE (refer to Section 2.2). Specifically, the standard deviations for code coverage by branch (according to KLEE), instruction (according to KLEE) and line (according to `gcov`) are 0.37%, 0.48%, and 1.56% respectively.

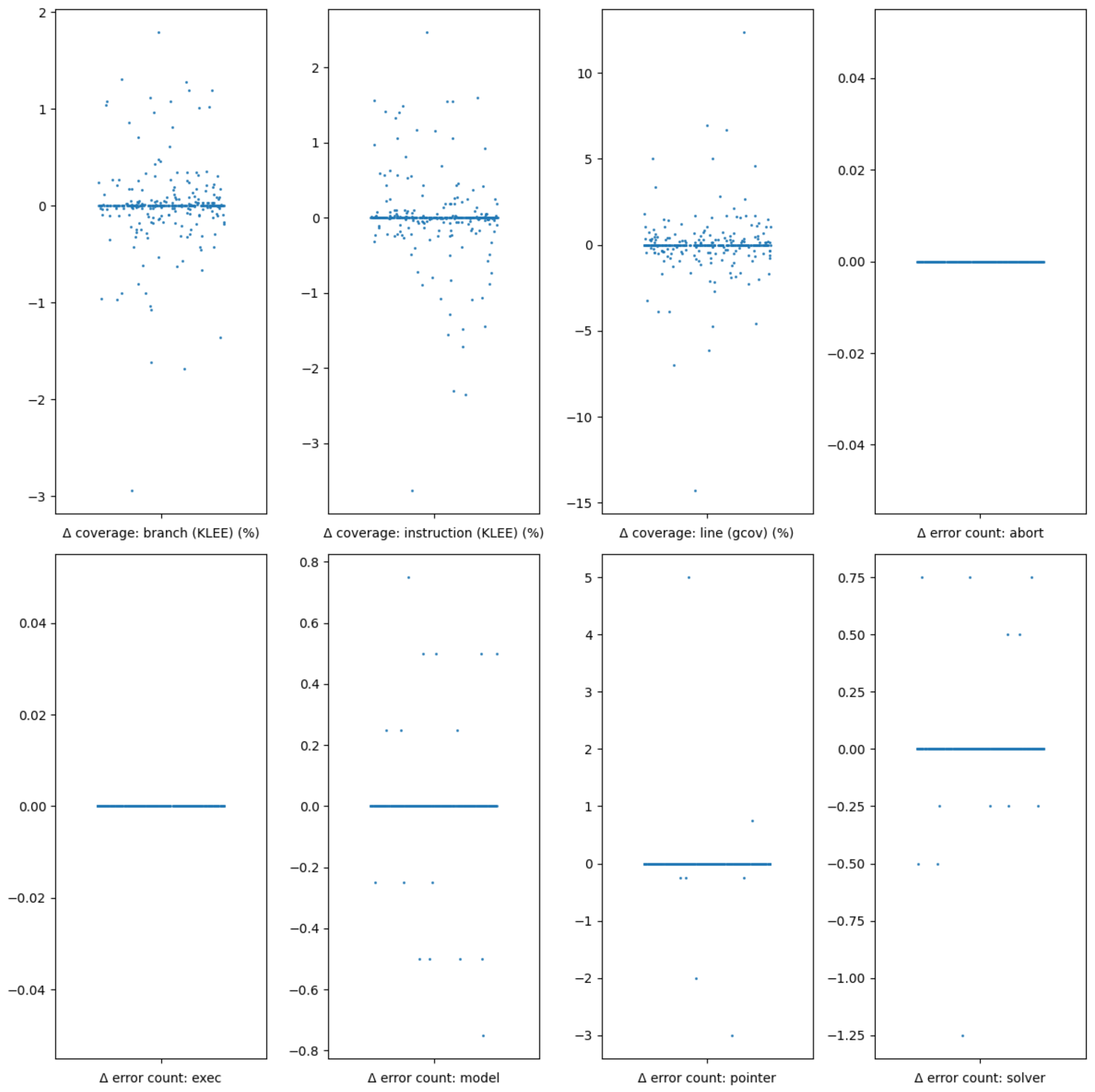

Figure 1: Spread of values normalized to the mean by util for coreutils 6.10 and a timeout of 60 minutes

Purely looking at the number of errors in each category is a very broad and imprecise way of measuring results. However, comparing and deduplicating errors found by a fuzzer is an art in itself, and simple approaches like comparing stack traces or the paths taken through software are insufficiently precise. The only accurate way to estimate the number (and severity) of errors is to manually look for the defect in the source code and compare the logic errors leading to the different findings. [10] This is a lengthy task and requires intimate knowledge of the software under test and was thus declared out of scope for this project.

4.1 Comparison to the Original Paper

Figure 2 is taken from the original KLEE paper [2] and shows the empirical cumulative distribution function (ECDF) of the coverage measured in their experiments. Figure 3 shows the same measurements from the four runs done with the same settings as in the original paper.

It is important to note that the settings proposed by the documentation [6] do not include failing system calls, thus need to be compared to "Base" (white) elements of the graph in Figure 2 only.

In general, Figures 2 and 3 do look reasonably similar, validating the approach taken in this paper.

However, there clearly is a difference visible between the two graphs, namely that the results I was able to achieve lag behind those reported in the original paper. Since discrete numbers from the original paper unfortunately are not available, and guessing results from a graph with mediocre resolution is error-prone, further numerical analysis of this difference is not performed. From looking at the variance in the four runs in Figure 3, it seems unlikely that the difference is purely statistical.

I do not have a definitive answer for the discrepancies, but the explanation might include differences in measurements, specifically in the calculation of executable lines of code (as discussed in Section 3.5), changes in the version of KLEE the original experiments were performed with and the current version of KLEE, and performance differences between the machines the experiments were run on. The latter would either require that the performance penalty incurred by using Docker is greater than the speedup gained by 15 years of hardware developments or be based on bottlenecks during the experiments, such as I/O bandwidth.

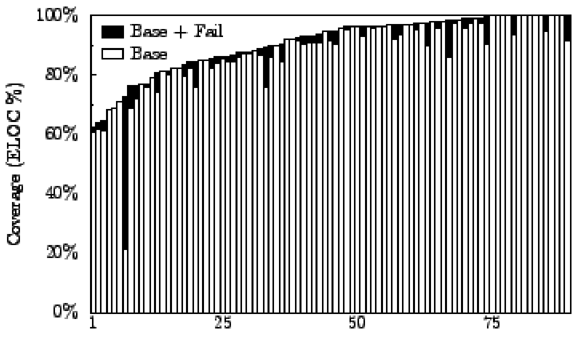

FIGURE 2: Coverage according to the original KLEE paper [2]

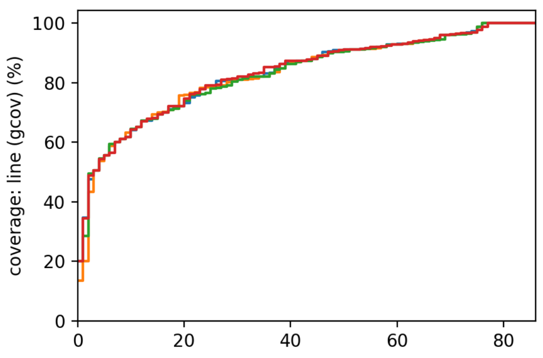

FIGURE 3: Coverage measured by `gcov` across four runs

4.2 Influence of Timeout

The performance drawbacks resp. benefits of my testing setup can be indirectly measured by changing the timeout passed to KLEE. This experiment was further inspired by the seminal work by Klees *et al.*, who collected a series of guidelines that should be followed to accurately measure the performance of a fuzzer. They propose that multiple runs should be performed to increase accuracy and found that "longer timeouts may be needed to paint a complete picture of an algorithm's performance" [10].

I chose three additional durations to conduct experiments at, with a large enough difference to ensure a complete picture of KLEE's performance across durations: 10 minutes, 6 hours and 24 hours.

4.2.1 Changes in Coverage

Figure 4 shows the ECDF of at least three runs across the different timeouts. Predictively, the coverage increases with increased timeouts. The difference between the timeouts does not seem too big, and this is confirmed by Figure 5.

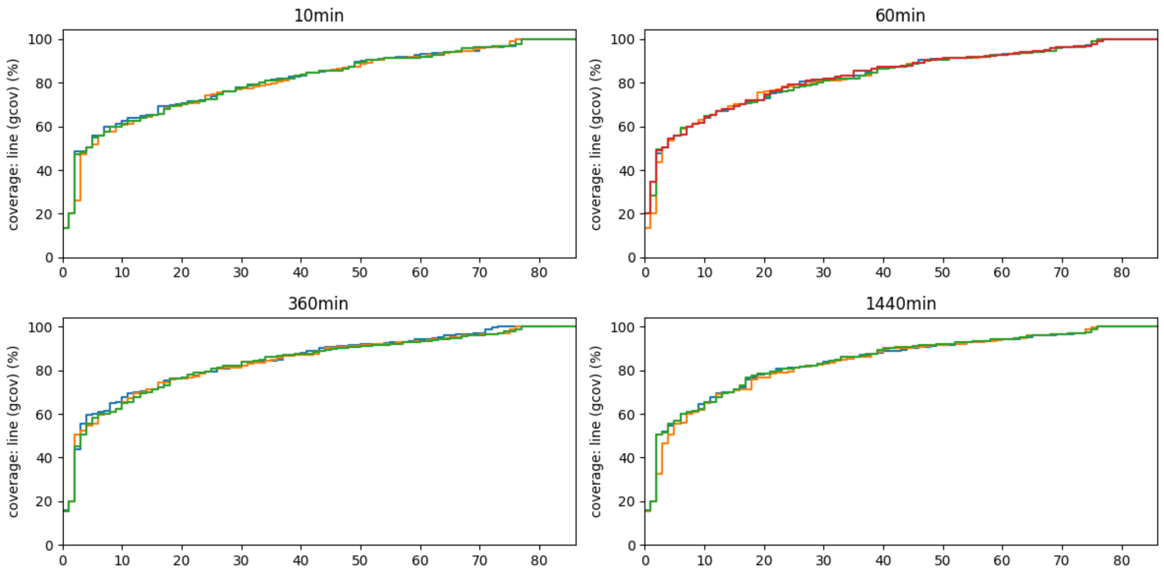

FIGURE 4: Coverage measured by `gcov` across different timeouts

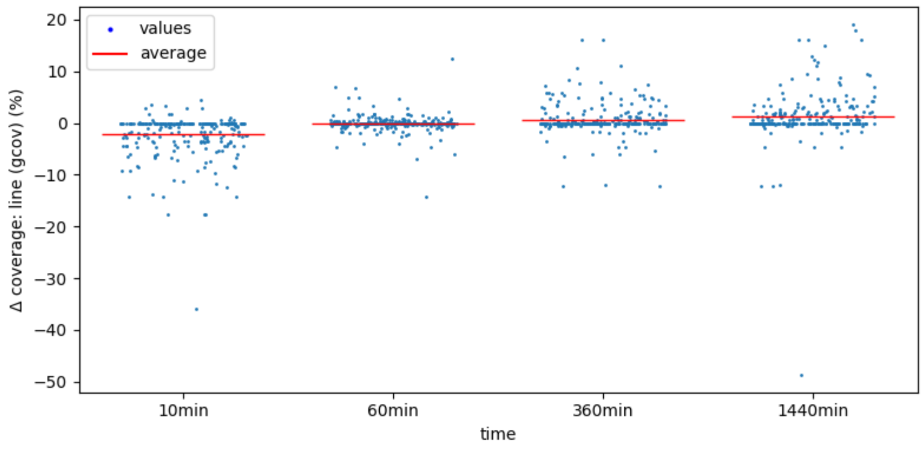

FIGURE 5: Spread of coverage by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

4.2.2 Changes in Number of Errors

Looking at changes in the number of errors in Figure 6 paints an incomplete picture: While the number of errors increases with the additional analysis time, looking at the changes in individual utils shows that there exist no utils where no issues were found with more than one hour of analysis but where lower timeouts produced findings. There are utils where runs with longer timeouts only sometimes found the same number of errors as the one hour runs, which can be explained by overall variance in the test setup. There are multiple instances of utils where additional analysis time produced significantly more errors. As an example, the average number of total errors found in the util `ptx` in one hour was 5. However, the runs with a timeout of 24 hours produced 16, 16, and 21 errors. For most utils however, as the figure suggests, the number of errors did not change based on timeouts.

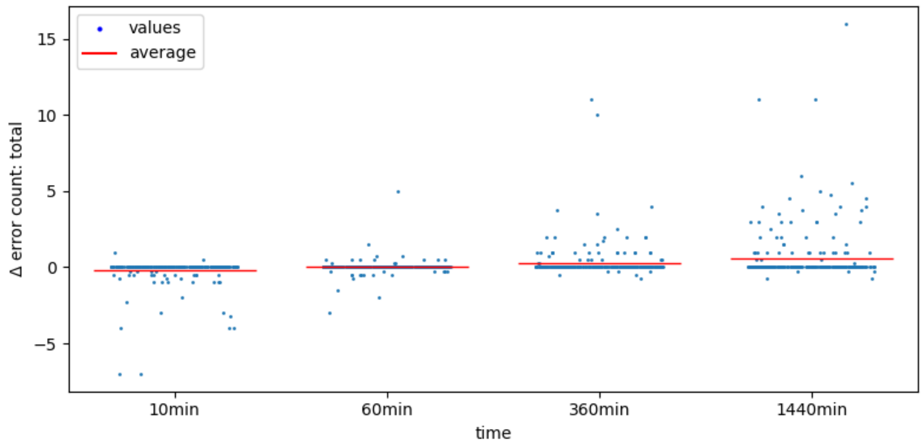

Figure 6: Spread of total number of errors found by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

5 Testing More Recent Versions of coreutils

As a second part, I wanted to see how the results from performing the same experiments on different versions of coreutils would behave. I chose two additional versions: 9.4, which is the current version at the time of the experiments, and 8.25, which was released exactly halfway between 6.10 and 9.4.

5.1 Differences in Testing Setup

The fundamental approach for the newer versions of coreutils remained the same: use old compilers on old systems where necessary and copy the binaries to KLEE's Docker image. Specifically, the following changes compared to the Dockerfile described in Section 3 were necessary:

- For coreutils 8.25 Ubuntu 16.04 was chosen as the base image according to the same logic described in Section 3.4. Similarly, Ubuntu 22.04 was chosen for coreutils 9.4, since KLEE requires at most LLVM 13 and Ubuntu 24.04 (which is the latest available LTS version available) no longer supports a version of LLVM compatible with KLEE.
- The step replacing parts of the source code of `sort.c`, as described in [6], was changed from replacing `(128 * 1024)` with `(1024)` (compared to from `(1024 * 1024)` to `1024` in coreutils 6.10) in both coreutils 8.25 and 9.4, since the source code was changed in the meantime.
- For coreutils 8.25, LLVM and clang need to be installed at version 3.5, for coreutils at version 13.
- For coreutils 8.25, pip needed to be upgraded to version 19, because otherwise a python incompatibility prevented the installation of WLLVM.

- For coreutils 8.25, the WLLVM version remained unchanged compared to 6.10, but version 1.3.1 was chosen for coreutils 9.4.
- The `configure` step when building both coreutils 8.25 and 9.4 requires an environment variable to run as root (which is the default user of Ubuntu's Docker images).
- Finally, the optimization flags for the LLVM bytecode generation step needed to be changed. According to the documentation, while just using `-O0` on its own is fine for LLVM 3.4 (which is what is used for coreutils 6.10), this is no longer recommended for more recent versions. [7], [11] However, I could not get the compiler to build coreutils 8.25 using the proposed solution of `-O1 -Xclang -disable-llvm-passes`. Specifically, I needed to remove the `-Xclang` argument to get the build stage to complete successfully.

5.2 Findings

I chose a fixed timeout of 60 minutes as a constant to compare the different versions of coreutils. I then executed three runs of each additional version.

5.2.1 Changes in Coverage

Figure 7 shows the coverage changes measured. The results indicate that the variance compared to version 6.10 predictively increases between the versions, as the code incrementally changes. Certain utils seem to become better penetrable by KLEE, while the coverage of others decreases, at times catastrophically. The overall average decreases across versions, which could be explained by increased complexity in the software as features are added.

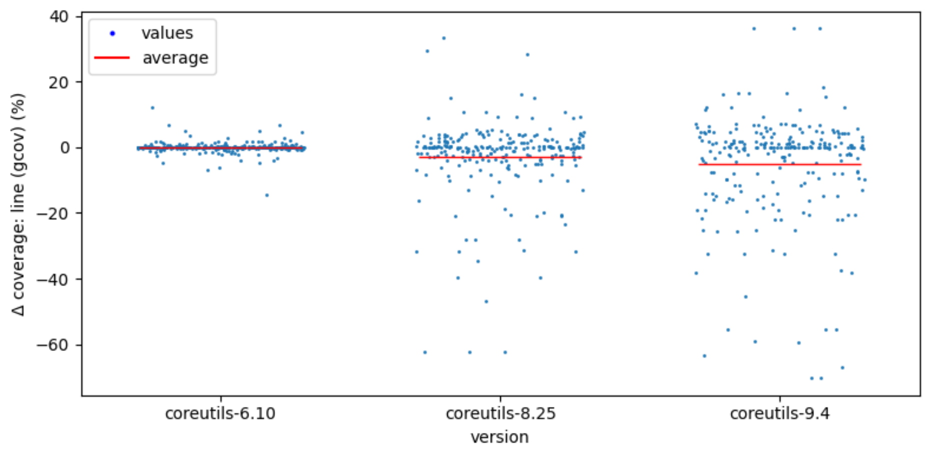

FIGURE 7: Spread of coverage by run, normalized to the mean of the util at version 6.10, across different versions

5.2.2 Changes in Number of Errors

The number of errors found by KLEE remains fairly constant across the software versions and seems to even increase slightly for version 9.4. This is surprising to me, as the number of software errors should ideally decrease with time as bugs are fixed. However, this effect might be mitigated by new code adding new features or adapting the software to a changing environment introducing new errors. It might also show that running off-the-shelf fuzzing tools can be a path to finding undiscovered errors in software systems.

However, this speculation is difficult to maintain in the face of my inability to crash any util by manually replaying the inputs created by `ktest-tool` for a sample of errors reported on version 9.4. This would indicate that many (if not all) of the reported errors are in fact false positives. However, to draw definitive conclusions about this, one would need to go through every error reported and manually check them. And, as argued in Section 4, this could not be done as part of this project.

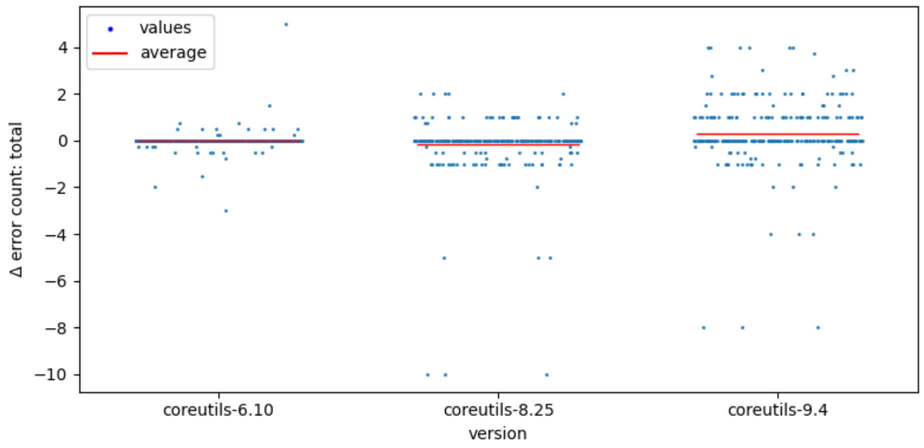

FIGURE 8: Spread of total number of errors found by run, normalized to the mean of the util at version 6.10, across different versions

6 Discussion

6.1 Research Questions

Section 3 showed that it is possible to run the current version of KLEE on coreutils 6.10, even though it requires multiple assembly steps across different Docker images to resolve version incompatibilities. The original paper provided two forms of measurements: coverage as reported by `gcov` on executable lines of code and number of errors. Examining the specific errors found by KLEE is too complex and big a task to be included in this project. In the face of this, I opted to using the number of errors as an easier to calculate if inaccurate proxy.

Section 4 goes into further detail on the variance between test runs. It then continues by contrasting the results reported by the original KLEE paper with the results measured in the experiments done during this project. For code coverage, my test results were similar to what was reported by Cadar *et al.*, validating the experiments. However, there still remain unexplained minor differences. Finally, the influence of different timeouts on the results is examined. Additional analysis time predictively leads to an increased coverage and number of errors found.

Section 5 discusses the results of the same experiment on different versions of coreutils. It introduces the changes necessary to run the test suite built in this project on versions 8.25 and 9.4. The results show a slightly decreasing code coverage with increased version number and a steady number of errors found. However, it also discusses how those errors might be largely false positives.

6.2 Produced Artifacts

6.3 Future Work

Bibliography

[1] "KLEE symbolic execution engine." (2024), [Online]. Available: `https://klee.github.io` (visited on Jan. 24, 2024).

[2] 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.

[3] V. Huber, "Challenges and mitigation strategies in symbolic execution based fuzzing through the lens of survey papers," Dec. 2023. [Online]. Available: `https://github.com/riesentoaster/review-symbolic-execution-in-fuzzing/releases/download/v1.0/Huber-Valentin-Challenges-and-Mitigation-Strategies-in-Symbolic-Execution-Based-Fuzzing-Through-the-Lens-of-Survey-Papers.pdf`.

[4] S. Flum and V. Huber, "Ghidrion: A ghidra plugin to support symbolic execution," Bachelor's Thesis, Zürich University of Applied Science — Institute of Applied Information Technology, Jun. 2023. [Online]. Available: `https://valentinhuber.me/assets/ghidrion.pdf`.

[5] "The simple theorem prover." (2024), [Online]. Available: `http://stp.github.io` (visited on Jan. 24, 2024).

[6] "OSDI'08 coreutils experiments." (2024), [Online]. Available: `https://klee.github.io/docs/coreutils-experiments/` (visited on Jan. 24, 2024).

[7] "Tutorial on how to use KLEE to test GNU coreutils." (2024), [Online]. Available: `https://klee.github.io/tutorials/testing-coreutils/` (visited on Jan. 24, 2024).

[8] Y. Li, S. Ji, Y. Chen, *et al.*, "UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 2777–2794, ISBN: 978-1-939133-24-3. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei`.

[9] "Whole program llvm." (2024), [Online]. Available: `https://github.com/travitch/whole-program-llvm` (visited on Jan. 24, 2024).

[10] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138, ISBN: 9781450356930. DOI: `10.1145/3243734.3243804`. [Online]. Available: `https://doi.org/10.1145/3243734.3243804`.

[11] "KLEE: -O0 is not a recommended option for clang." (2024), [Online]. Available: `https://github.com/klee/klee/issues/902` (visited on Jan. 24, 2024).

Appendix

1 Building coreutils 6.10 on KLEE's Docker image

1.1 Error

```
depbase=`echo freadahead.o | sed 's|[^/]*$|.deps/&|;s|\.o$||'`;\
gcc  -I. -I../../lib      -g -O2 -MT freadahead.o -MD -MP -MF $depbase.Tpo -c -o freadahead.o ../../lib/freadahead.c &&\
mv -f $depbase.Tpo $depbase.Po
../../lib/freadahead.c: In function 'freadahead':
../../lib/freadahead.c:64:3: error: #error "Please port gnulib freadahead.c to your platform! Look at the definition of fflush, fread on your system, then report this to bug-gnulib."
```

```
   64 |  #error "Please port gnulib freadahead.c to your platform! Look at the definition of fflush, fread on your system, then report this to bug-gnulib."
      |    ^~~~~
make[2]: *** [Makefile:1245: freadahead.o] Error 1
make[2]: Leaving directory '/home/klee/coreutils-6.10/obj-llvm/lib'
make[1]: *** [Makefile:905: all] Error 2
make[1]: Leaving directory '/home/klee/coreutils-6.10/obj-llvm/lib'
make: *** [Makefile:769: all-recursive] Error 1
```

1.2 freadahead.c

The license has been cut for brevity's sake.

```
#include <config.h>

/* Specification.  */
#include "freadahead.h"

size_t
freadahead (FILE *fp)
{
#if defined _IO_ferror_unlocked     /* GNU libc, BeOS */
  if (fp->_IO_write_ptr > fp->_IO_write_base)
    return 0;
  return fp->_IO_read_end - fp->_IO_read_ptr;
#elif defined __sferror             /* FreeBSD, NetBSD, OpenBSD, MacOS X, Cygwin */
  if ((fp->_flags & __SWR) != 0 || fp->_r < 0)
    return 0;
  return fp->_r;
#elif defined _IOERR                /* AIX, HP-UX, IRIX, OSF/1, Solaris, mingw */
# if defined __sun && defined _LP64 /* Solaris/{SPARC,AMD64} 64-bit */
#  define fp_ ((struct { unsigned char *_ptr; \
                         unsigned char *_base; \
                         unsigned char *_end; \
                         long _cnt; \
                         int _file; \
                         unsigned int _flag; \
                       } *) fp)
  if ((fp_->_flag & _IOWRT) != 0)
    return 0;
  return fp_->_cnt;
# else
  if ((fp->_flag & _IOWRT) != 0)
    return 0;
  return fp->_cnt;
# endif
#elif defined __UCLIBC__            /* uClibc */
# ifdef __STDIO_BUFFERS
  if (fp->__modeflags & __FLAG_WRITING)
    return 0;
  return fp->__bufread - fp->__bufpos;
# else
  return 0;
# endif
#elif defined __QNX__               /* QNX */
  if ((fp->_Mode & 0x2000 /* _MWRITE */) != 0)
    return 0;
  /* fp->_Buf <= fp->_Next <= fp->_Rend */
  return fp->_Rend - fp->_Next;
#else
 #error "Please port gnulib freadahead.c to your platform! Look at the definition of fflush, fread on your system, then report this to bug-gnulib."
#endif
}
```