Running KLEE on GNU coreutils

—

Report

Valentin Huber

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

February 13, 2024

Abstract

In 2008, Cadar *et al.* presented KLEE: a symbolic execution based fuzzer. In their report [1], they evaluate the performance of KLEE by running it against 89 programs from the GNU coreutils suite. In this project, I repeated those experiments and attempted to reproduce the results reported, along with further experiments. Specifically — after a brief introduction to KLEE and symbolic execution based fuzzing in general — I examined the statistical variance introduced by the inherent randomness of KLEE's design, and how changing the timeout passed to KLEE changes the measured results. I further ran the same experiments on two additional more recent versions of coreutils, to see how KLEE performs on current software, and to get a sense of how software evolves over time. I then discuss how the results obtained in these experiments are analyzed and visualized. Finally, I reflect on this project, including a presentation of the artefacts produced and ideas for future work.

Contents

1 Introduction

KLEE [2] is an open source advanced fuzzing platform based on symbolic execution. It was introduced in the seminal paper entitled "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 against a number of programs. The most prominent of these is the GNU coreutils suite, in which ten fatal bugs were found.

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

1. Reproducing the original paper (see Section 3)
   (a) Can the current version of KLEE run on coreutils version 6.10 (as tested in the original paper)?
   (b) Can the same metrics be measured as in the original paper?
2. Examining the statistical distribution of the results and comparing them with the results published in the original paper (see Section 4)
   - How does the non-determinism in KLEE affect the variance of the results between different test runs?
   - How do the measured metrics compare to what was published 15 years ago?
3. Investigating the influence of different fuzzing timeouts on the results (see Section 5)
   (a) How do different fuzzing timeouts affect the results?
4. Testing more recent versions of coreutils (see Section 6)
   (a) What needs to be changed in the test setup to test more recent versions of coreutils?
   (b) How do the results of testing different versions of coreutils differ?

All experiments were run on a virtualized server with the following specifications AMD EPYC 7713 64C 225W 2.0GHz processor, 1 TiB RAM, 2x 25GiB/s Ethernet. The hardware was kindly provided by the Institute of Applied Information Technology at Zürich University of Applied Sciences [4].

2 Background

The following is a brief explanation of the application of symbolic execution in fuzzing. For more extensive background, I refer to some of my previous work [3], [5].

Recall that KLEE is an open source fuzzer based on symbolic execution. It takes LLVM bytecode from the program under test (PUT) as input and performs its analysis on it. KLEE then outputs some statistics about the execution, inputs to the PUT that crash it, and inputs that, when executed, cover all the branches KLEE 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 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 relation $\phi_k = \phi_i + \phi_j$. Conditional jumps are mapped to conditions on these variables for both results of the condition, so the instruction `%isPositive = icmp sgt i32 %result, 0` would be represented by $\phi_k > 0$ and $\phi_k \leq 0$, respectively.

The set of all conditions along a given path through the PUT is called *path condition.* It can be passed to a Satisfiability Modulo Theory (SMT) solver (KLEE uses STP [6] by default), which will return values for all user input such that the PUT is forced down the exact path represented by the path condition.

This is the main advantage of symbolic execution based fuzzing over traditional fuzzing. By not using concrete values, but logical representations of user input, it essentially runs the PUT with all possible user inputs *simultaneously.* So if the solver returns that no input satisfies 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 into 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 will give a brief summary here:

- Environment interactions (such as file system interactions) are generally 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 using concrete values in the call. This abandons the claim to completeness that symbolic execution typically has, but is often the only feasible way to continue the analysis.

- The second major challenge of symbolic execution is what is known as *path explosion.* Because the number of program states grows exponentially with the number of instructions, it is infeasible to compute the entire state space for all but the simplest programs. KLEE deals with this by reducing the search space to actually executable instructions, using advanced data structures, and examining paths through the PUT sequentially, 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 chooses either the input that promises to increase coverage the most, or a random input to prevent execution from getting stuck in a particular subtree of the PUT.

- KLEE needs to model the entire memory of a process. This is straightforward as long as variables are used directly, but becomes a challenge when pointers are involved. This is especially true when the value of these pointers depends on user input, as this would require KLEE to model all possible addresses with all possible values, which would instantly explode the memory usage and the number of states to examine, and is therefore infeasible. KLEE deals with this by representing such pointer operations as array accesses, where the accessed object is copied as many times 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 runtime of the fuzzer. KLEE applies some advanced optimizations, such as query splitting and more general optimizations, or a cache of previous results, which often solve supersets of the query they are a solution to. Finally, KLEE defines a timeout, after which the solver is interrupted and the analysis is continued on another branch.

3 Reproducing the Original Paper

I'm basing my experiment setup on the original paper [1], the FAQs in the project documentation [7], and the tutorial on testing coreutils version 6.11 [8].

3.1 Project Setup

KLEE is a complex system with 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 rather in comparing different setups, this is a tradeoff worth making. Using Docker to evaluate fuzzer performance has been done before [9]. Finally, this makes complex build steps reproducible and serves as documentation.

3.2 Naïve Approach

Attempting to build coreutils 6.10 directly from the current version of KLEE's Docker image will result in an error: The Docker image is based on Ubuntu 22 (Jammy) and is no longer able to build coreutils 6.10 with the GNU Compiler (GCC). This is because the coreutils build system tries to detect what system it is running on, and the variable it relies on for detection is no longer defined. Specifically, the following check is performed in `freadahead.c`:

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

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

3.3 Using an Old Version of Ubuntu

One attempt to mitigate this problem 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 decided to take a different approach for two reasons:

1. Build systems are not my area of expertise, and I do not know how many other problems would arise once the first one was solved.
2. Changing code always carries the risk of introducing additional software bugs that would skew my results.

Therefore, I tried 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 that was available when coreutils 6.10 was current. This approach worked without any additional changes to the code or build system. The setup of the Docker image used to build coreutils can be seen in Listing 1.

3.4 Generating LLVM Bytecode Files

Building binaries is not enough, because KLEE does not take pure binaries as input, but requires LLVM bytecode. Compiling a normal `.c` file to LLVM can easily be done with `clang`. But again, coreutils use a complex build system, which means that in order to just use `clang`, one would have to deeply understand and modify it, with the drawbacks mentioned above.

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

This is where Whole Program LLVM (WLLVM) [10] comes in, a tool specifically designed to work with complex build systems while still producing LLVM bytecode as one of its outputs. It accomplishes this by injecting its compiler into the build system. The compiler creates executable binaries but also injects LLVM bytecode into a dedicated section of the object files. In a second step, these are extracted and linked together to create 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, since newer versions require a version of Python that is not available on Ubuntu

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
```

14.04. To create proper input files for KLEE, I added two options to reduce warnings (`--build`) and to disable premature optimizations according to the KLEE documentation (`CFLAGS`) [8].

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 with GCC and tell the compiler to add coverage measurement instrumentation. A note document (`<executable-name>.gcno`) is created along with the binaries. When the binary is executed, the added instrumentation records the path taken through the code and together with information from the notes file, stores its results in a coverage data file (`<executable-name>.gcda`). This file can then be parsed with `gcov` to get human-readable coverage data.

With this step, however, I ran into the same problem as before: Recent versions of GCC no longer build coreutils 6.10. I took 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 these instructions are also included in the measurements. This was done according to the instructions in the FAQ [7].
- The original paper mentions that coverage is measured only on executable lines of code. Specifically, Section 5.1 of the original paper states

  > “We measure size in terms of executable lines of code (ELOC) by counting the total number of executable lines in the final executable after global optimization, which eliminates uncalled functions and other dead code.” [1]

  It remains unclear how Cadar *et al.* calculated the executable lines, as this is not a trivial task. I enabled global optimization (`-O2`), but this may still result in a considerable underestimation of coverage.

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

```dockerfile
# ========================================
# 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

```dockerfile
# ========================================
# 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}"
```

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 [7]. Then the bytecode files from the step described 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 a complex process itself, and is performed by running a shell script (`analyze.sh`). This step is explained in Section 3.7. The analysis script is copied to the image and executed when the container is started. To pass certain settings to the analysis step, environment variables are used that 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 the Docker image created in the previous steps is started, 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 is parsed, including the settings passed. 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 additional metrics to be collected based on the output of a previous fuzzing run without having to perform additional, computationally expensive analysis.

2. To run KLEE, the analyst must pass arguments specifying the size and number of inputs and input files to test. For most coreutils, this is the same, but (as mentioned in Section 5.2 of the original paper [1] and explained in the FAQs [7]) some utils require different settings to achieve adequate coverage. The analysis script assembles the command to run KLEE, including the constant settings, the util-dependent settings, and the timeout set in the script arguments.
3. Then the actual fuzzing is performed.
4. KLEE's output is examined in several ways:
   (a) For each error found, human-readable output is generated 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 to `klee-replay`, pointed at the binary instrumented with coverage-gathering instructions. This ensures that every instruction that KLEE analyzes during its fuzzing is actually executed and thus recorded in the coverage results. Since the instrumented binaries were not compiled on the same system as they are executed, the environment variables `GCOV_PREFIX` and `GCOV_PREFIX_STRIP` must be set according to the changed environment.
5. Finally, certain large output text files are compressed to minimize disk usage.

3.8 Extracting Human-Readable Coverage Data

The final step is to feed the output of the binaries instrumented to collect coverage metrics back into `gcov`. Unfortunately, the format of these output files has 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 moved back to the Docker image that created them, where an obviously compatible version of `gcov` is available.

3.9 Gathered Metrics

The following metrics were collected for each run on each util of the experiment described above:

- The name of the util, including version and run name
- 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 occur on illegal instructions and external calls with a symbolic `errno` call
  - `model` errors, which occur when concretizing a symbolic size
  - `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` for all except three utils: `[`, `ginstall`, and `md5sum` did not produce a `.gcno` file, preventing coverage gathering.

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

[2] KLEE reports a number of additional metrics such as 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 Experiments Results to the Original Paper

The following sections discuss only a small subset of the plots generated based on the metrics collected during the experiments. Summary plots showing the spread and the empirical cumulative distribution function (ECDF) of all measurements across timeouts and versions, as well as plots showing the results of individual utils, are available in the repository of this experiment [11].

Figure 1 shows the spread of measurements between four runs of the experiment with the same parameters. The non-deterministic nature of the results is due to the inherent non-determinism of the fuzzing process in KLEE (see 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.

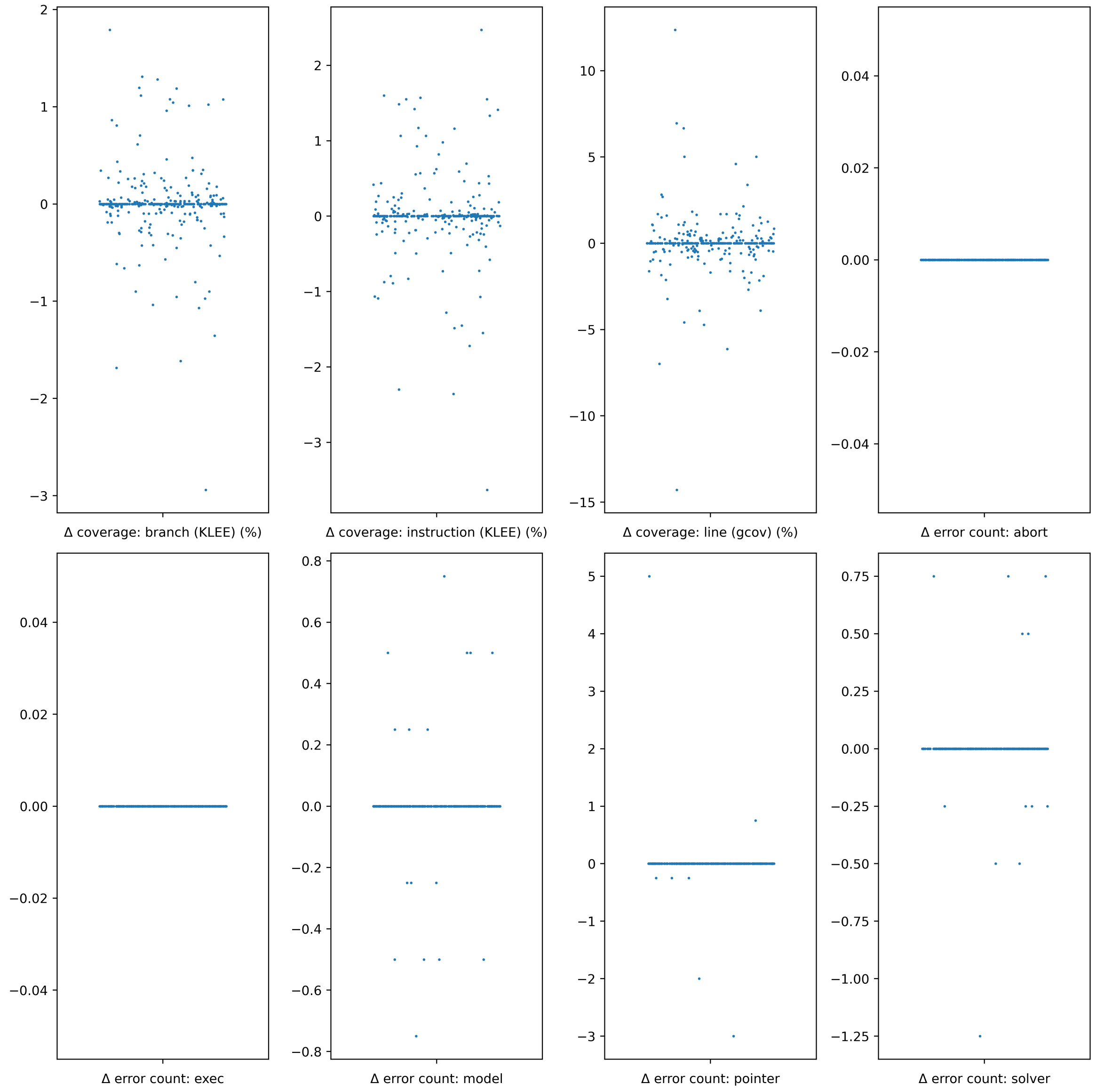

Figure 1: Spread of values normalized to the mean by util for coreutils 6.10 and a timeout of 60 minutes

Looking only at the *number* of defects in each category is a very broad and imprecise way to measure results. However, comparing and deduplicating defects found by a fuzzer is an art in itself, and simple approaches such as comparing stack traces or paths taken through software are not precise enough. The only accurate way to estimate the number (and severity) of defects is to manually search for the defect in the source code and compare the logic errors that lead to the different results. [12] This is a laborious task and requires intimate knowledge of the software under test, and was therefore declared out of scope for this project.

4.1 Comparison to the Original Paper

Figure 2 is taken from the original KLEE paper [1] and shows the ECDF of the coverage measured in their experiments. Figure 3 shows the same measurements from the four runs made with the same settings as in the original paper.

It is important to note that the settings suggested by the documentation [7] do not include failed system calls, and thus need only be compared to the "Base" (white) elements of the graph in Figure 2.

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

However, there is clearly a difference 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 are not available, and since guessing results from a graph with mediocre resolution is error-prone, no further numerical analysis of this difference is performed. 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 could include differences in the measurements, especially in the calculation of executable lines of code (as discussed in Section 3.5), changes in the version of KLEE with which the original experiments were run and the current version of KLEE, and performance differences between the machines on which the experiments were run. The latter would either require that the performance penalty from using Docker be greater than the speedup gained from 15 years of hardware development, or be based on bottlenecks during the experiments, such as I/O bandwidth.

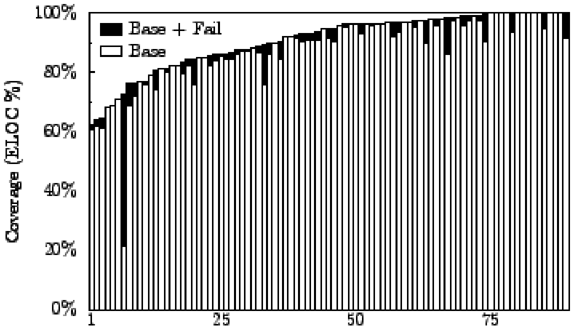

Figure 2: Coverage according to the original KLEE paper [1]

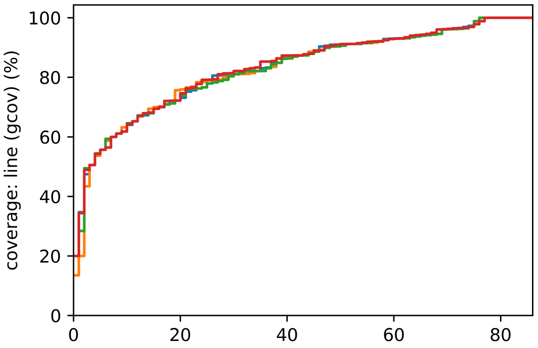

Figure 3: Coverage measured by `gcov` across four runs

5 Influence of Timeout

The performance drawbacks or benefits of my test setup can be measured indirectly by changing the timeout passed to KLEE. This experiment was also inspired by the seminal work of Klees *et al.*, who collected a set of guidelines that should be followed to accurately measure the performance of a fuzzer.

They suggest 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” [12].

I chose three additional durations to run 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.

5.1 Changes in Coverage

Figure 4 shows the ECDF of at least three runs across the different timeouts. Predictably, the coverage increases with increasing timeouts. The difference between the timeouts does not seem to be large, and this is confirmed by Figure 5.

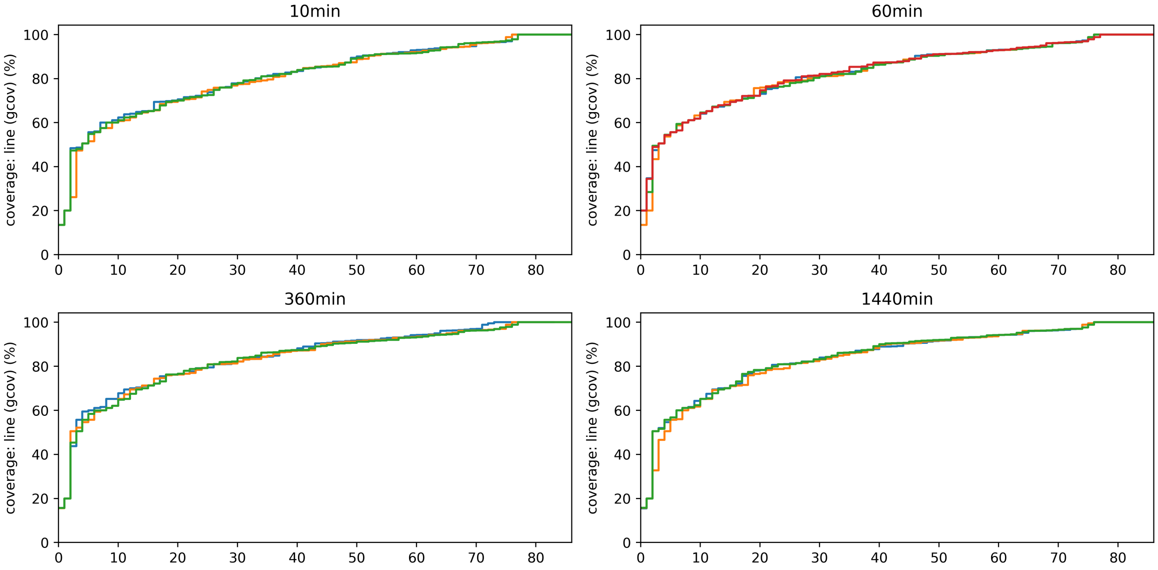

Figure 4: Coverage measured by gcov across different timeouts

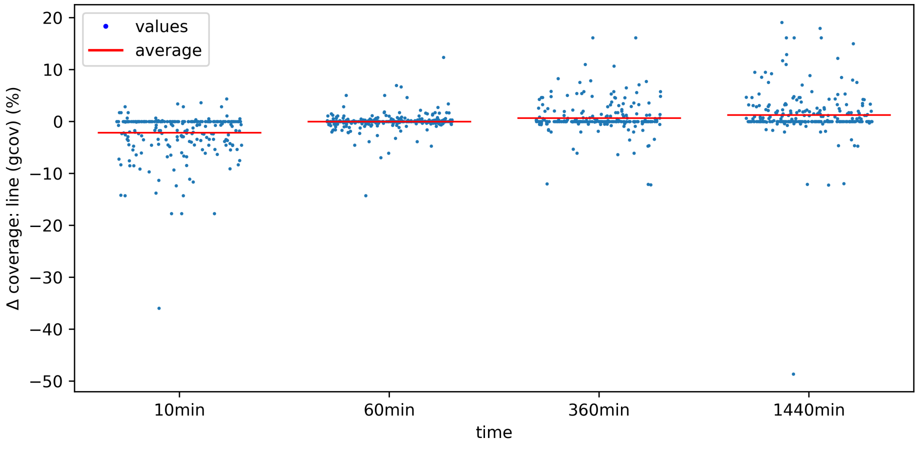

Figure 5: Spread of coverage by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

While the average coverage measured by gcov increases slightly as the timeout increases, this is not necessarily true for individual utils. Their behavior with respect to the number of distinct result values they produce can be broadly grouped as follows:

- 34 utils showed a fixed, constant coverage across all runs at all timeouts.
- 10 utils showed four or fewer discrete values that the measurements jumped between.
- The remaining utils showed a result on a more continuous scale.

The trends they show across different timeouts vary widely. The run results of 12 utils contain values that are lower than any value measured with a lower timeout. The same is true for the averages of 23 utils.

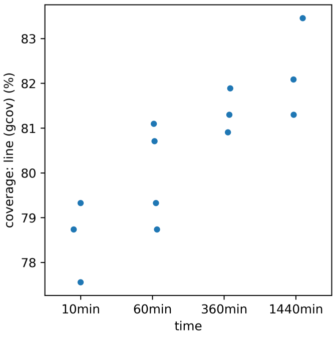

Figure 6: Coverage across times on `csplit`

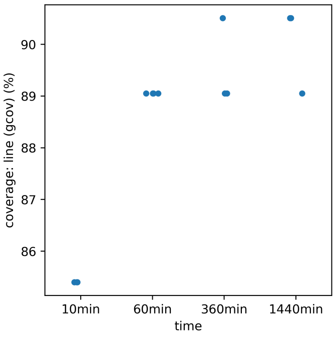

Figure 7: Coverage across times on `expand`

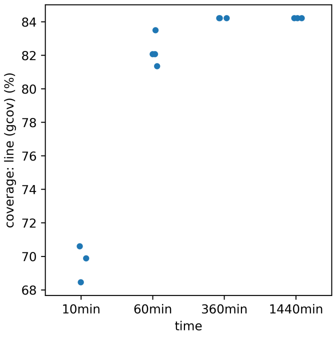

Figure 8: Coverage across times on `du`

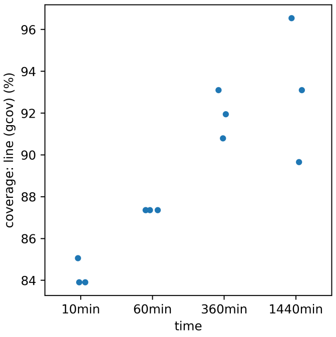

Figure 9: Coverage across times on `chown`

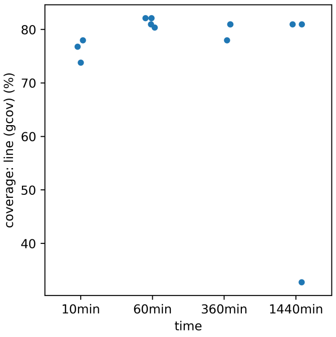

Figure 10: Coverage across times on `dircolors`

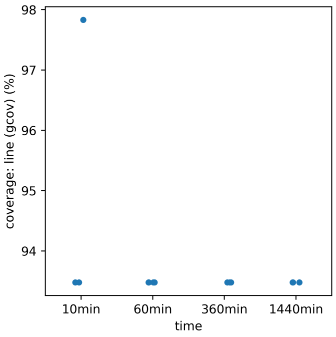

Figure 11: Coverage across times on `mkfifo`

I have selected a few utils to discuss here to give an introduction to the types of different results produced.

- Figure 6 represents the norm and what would be expected. As the analysis time increases, the minimum, maximum, and average of the results increase.
- Figure 7 is an example of what I called discrete values above. It still represents a strict increase in value over time.
- Figure 8 shows an effect that can be observed for multiple measurements and multiple utils: As time increases, a threshold is reached and further increases in time no longer increase the recorded value.

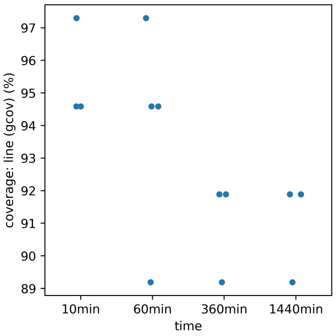

FIGURE 12: Coverage across times on `sleep`

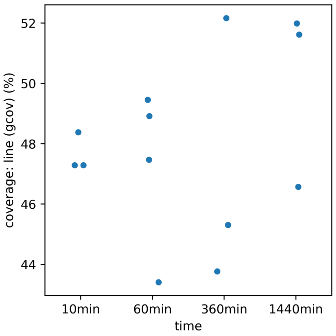

FIGURE 13: Coverage across times on `sort`

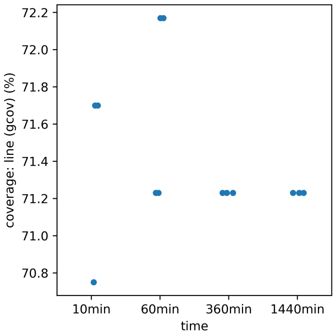

FIGURE 14: Coverage across times on `tac`

- Figure 9 shows a util where the minimum value no longer strictly increases. This is likely due to statistical randomness and is not sufficient to reject the hypothesis of a correlation between timeout and coverage without additional data points. It also means that statistical summaries retain significant error bars and utils need to be considered individually.
- Figure 10 and Figure 11 show two utils that produced unexpected outliers. While a single run for `dircolors` with a timeout of 24 hours suddenly produces only half the coverage, a single run with only 10 minutes for `mkfifo` managed to cover about two thirds of the missing code. The latter can be explained by the randomness inherent in KLEE's input selector (see Section 2.2).
- Figure 12 shows a counterintuitive result: With increasing time, the achieved coverage seems to decrease. I have no explanation for this effect, but additional data points might reduce the likelihood of this occurring due to statistical randomness.
- Figure 13 seems to show no correlation between the lines covered by KLEE's analysis and the timeout given to KLEE. As above, this may be due to statistical randomness.
- Figure 14 finally shows a similar apparent non-dependency between time and coverage, but with discrete values and an apparent approach to a number other than the maximum. Again, this could be due to statistical randomness.

5.2 Changes in Number of Errors

Looking at the changes in the number of errors in Figure 15 paints an incomplete picture: While the number of errors increases with additional analysis time, looking at the changes in individual utils shows that there are no utils where no problems were found with more than one hour of analysis, but where lower timeouts produced at least one finding.

The graphs of the number of each error over time for each util can be found in the repository [11]. Their behavior can be categorized as follows:

- KLEE found a constant number of errors in most utils.
  - For most of them, this number was zero.
  - In `cat`, `csplit`, `dd`, `md5sum`, `mkdir`, `mkfifo`, `mknod`, `od`, `seq`, `split`, and `unexpand` KLEE found a single pointer error in all runs.
  - In `ls` it found a single exec type error in all runs.

- For 19 utils, strict increases in the number of errors across utils were recorded. This means that the number of errors found at a timeout is equal to or greater than the number found at a shorter timeout. For example, the average number of total errors found in the `ptx` util in one hour was 5. However, the 24-hour runs produced 16, 16, and 21 errors.
- The three remaining utils show a different graph:
  - `expr` produced a query timeout (i.e., a solver-type error) in one run only with a timeout of one hour. Otherwise, the total number of errors found was a constant zero.
  - `shred` continuously produced one exec type error, a strict increase between zero and one model and pointer type error each, and a strict increase for solver type errors. However, due to the distribution of these across runs, the total is not a strict increase, because the only run that produced only one error in total has a timeout of one hour.
  - `chgrp` only showed errors of type model. All four runs with a timeout of one hour produced the error, but only two of three runs with a timeout of six hours and one of three runs with a timeout of 24 hours produced the same error.

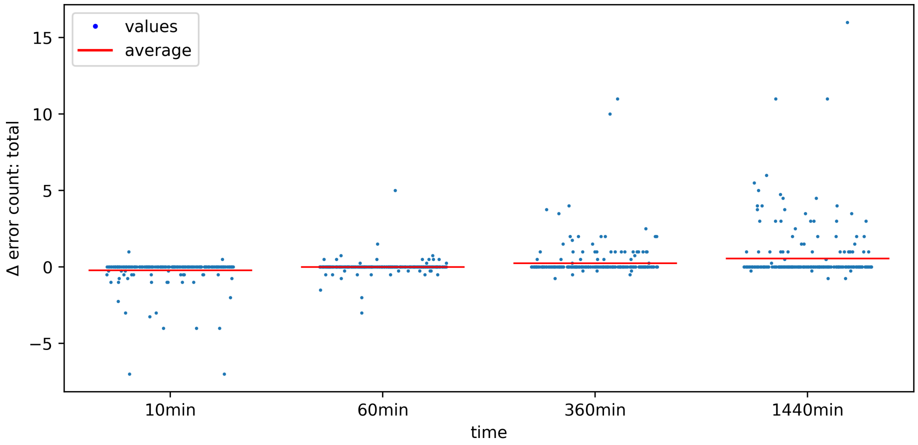

FIGURE 15: Spread of total number of errors found by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

6 Testing More Recent Versions of coreutils

As a second part, I wanted to see how the results of 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.

6.1 Differences in Test Setup

The basic 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 were required from the Dockerfile described in Section 3:

- 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 because KLEE requires at most LLVM 13 and Ubuntu 24.04 (which is the latest available LTS version) no longer supports a KLEE-compatible version of LLVM.

- The step of replacing parts of the source code of `sort.c`, as described in [7], has been changed in both coreutils 8.25 and 9.4 from replacing `(128 * 1024)` with `(1024)` (as opposed to replacing `(1024 * 1024)` with `1024` in coreutils 6.10), since the source code has been changed in the meantime.
- For coreutils 8.25, LLVM and clang need to be installed at version 3.5, for coreutils 13.
- For coreutils 8.25, pip must be upgraded to version 19, otherwise a Python incompatibility will prevent the installation of WLLVM.
- For coreutils 8.25, the WLLVM version remained unchanged from 6.10, but for coreutils 9.4, version 1.3.1 was chosen.
- The `configure` step in building both coreutils 8.25 and 9.4 requires an environment variable to run as root (which is the default user in 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` by itself is fine for LLVM 3.4 (which is what is used for coreutils 6.10), this is no longer recommended for more recent versions. [8], [13] However, I could not get the compiler to build coreutils 8.25 using the suggested solution of `-O1 -Xclang -disable-llvm-passes`. Specifically, I had to remove the `-Xclang` argument to get the build to complete successfully.

Despite all adjustments as described above, certain measurements could not be made:

- `hostname` and `setuidgid` do not seem to exist anymore in coreutils 8.25 and 9.4, so no measurements could be performed on them.
- `base64` did not produce a `.gcno` file on coreutils 8.25 and 9.4, thus making coverage information according to `gcov` unavailable.
- `cksum` and `sum` did not produce `.gcno` files in coreutils 9.4.
- `wc` did not produce coverage based on `gcov` in coreutils 9.4 for an unspecified reason.

6.2 Findings

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

6.2.1 Changes in Coverage

Figure 16 shows the measured changes in coverage. The results show that the variance compared to version 6.10 predictably increases between versions as the code incrementally changes. Certain utils seem to become more penetrable by KLEE, while the coverage of others decreases, at times catastrophically. The overall average decreases between versions, which might be explained by the increasing complexity of the software as features are added.

6.2.2 Changes in the Number of Errors

The number of errors found by KLEE remains fairly constant across software versions and even seems to increase slightly for version 9.4. This is surprising to me, as the number of software defects should ideally decrease over 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 that introduces new bugs. It

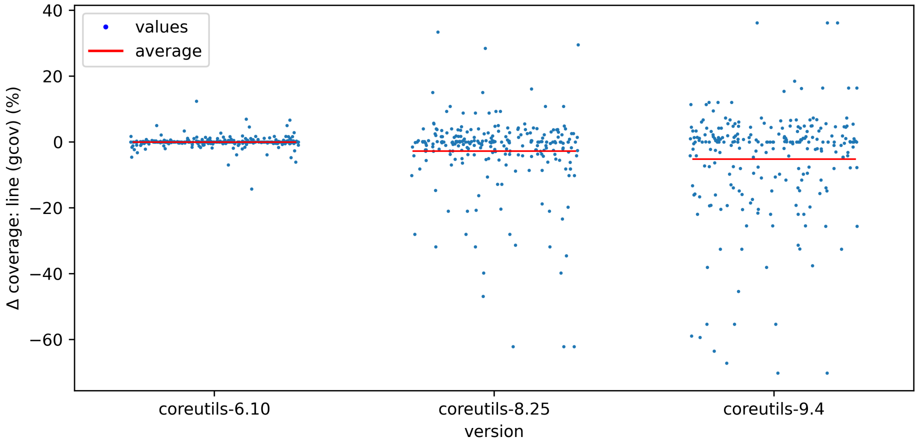

FIGURE 16: Spread of coverage by run, normalized to the mean of the util at version 6.10, across different versions

may also show that running standard fuzzing tools can be a way to find undiscovered bugs 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 generated by `ktest-tool` for a sample of errors reported in version 9.4. This may indicate that reported errors need to be examined and inputs modified further by hand to actually find crashes, or it may indicate that many (if not all) of the reported errors are in fact false positives. However, there is no way to draw definitive conclusions about this other than by manually examining each individual error. And, as explained in Section 5, this could not be done as part of this project.

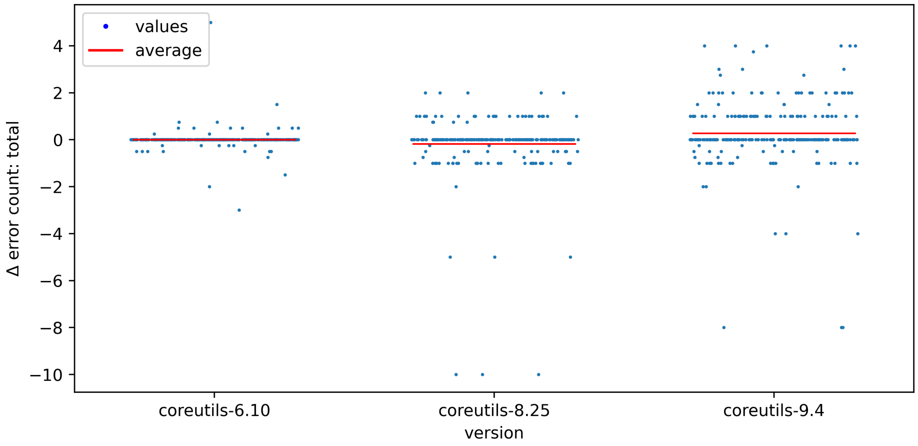

FIGURE 17: Spread of total number of errors found by run, normalized to the mean of the util at version 6.10, across different versions

7 Discussion

In this project, I successfully recreated the experiments outlined in the seminal work by Cadar *et al.* in 2008. I presented the project setup and specific results along possible interpretations. In this section, I will discuss the results in a greater context, what questions could be answered and what work remains.

7.1 Research Questions

Section 1 introduced a series of research questions to be answered in this report. Here, I summarize the answers discussed in Sections 3, 4, 5, and 6.

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. With additional steps during each run, `gcov` can be employed to measure coverage. Since no documentation is available on how exactly the authors of the original paper measured executable lines of code, comparing these measurements might not be valid.

Section 4 discusses the variance between test runs in more detail. It then contrasts the results reported in the original KLEE paper with the results measured in the experiments conducted during this project. For code coverage, my test results were similar to those reported by Cadar *et al.*, thus validating the experiments. However, there are still small unexplained differences.

Section 5 examines the influence of different timeouts on the results. Additional analysis time predictively increases coverage and the number of bugs found, but certain utils behave unpredictably when considered individually. Additional experimentation may reduce the statistical randomness and provide more clarity.

Section 6 discusses the results of the same experiment on different versions of coreutils. It presents 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 increasing version number, and a steady number of errors found. However, it is also discussed how these bugs may be largely false positives.

7.2 Produced Artifacts

Since all code used for and all raw data produced in this project is publicly available [11], it can be used as a basis for further experiments and analysis. Specifically, the following artifacts have been produced:

1. A framework for performing KLEE-based analysis on coreutils. This includes:
   - Dockerfiles for three different versions of coreutils (6.10, 8.25, and 9.4) that document the steps required to build all the artifacts needed to run KLEE and collect comprehensive metrics.
   - `analyze.sh`, a shell script responsible for the actual analysis, including argument assembly, input and output handling, and metrics collection.
   - `run-suite.py`, a Python script to efficiently run experiments on all (or a subset of) coreutils, including handling arguments and output files.
2. Results of 19 runs of KLEE on all coreutils, totaling almost 10000 CPU hours.
3. Extensive scripts analyzing the results of the above framework in `analyze.ipynb` and the corresponding plots:
   - For each measurement spread (like Figures 5, 15, 16, and 17) and ECDF (like Figures 3 and 4) plots, showing the evolution of the value over KLEE timeouts and coreutils versions.

- For each measurement and util plots showing the results (similar to Figures 6, 7, 8, 9, 10, 11, 12, 13, and 14).

7.3 Future Work

Future work can be broadly divided into three parts:

First, additional runs with the same setup will improve the reliability of the measured results by reducing statistical randomness. This is discussed further in Section 5.

Second, the results could be improved by ensuring that both the project setup (such as compilation arguments) and the way the measurements are taken are exactly the same as in the original paper. The latter is especially important since the way coverage is measured with `gcov` is not described precisely enough in the original paper (see Section 3.5 for details).

Finally, the errors reported in the different runs need to be investigated further to allow further validation of the experiment setup by checking if the same software errors were found by the original authors and in my experiments. Furthermore, it could provide additional insight into the comparison of different coreutils versions. To start with, one could analyze two consecutive versions and check if bugs mentioned in the patch notes are also found in KLEE's output.

Bibliography

[1] 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.

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

[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] "Institute of applied information technology." (2024), [Online]. Available: `https://www.zhaw.ch/en/engineering/institutes-centres/init/` (visited on Jan. 24, 2024).

[5] 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`.

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

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

[8] "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).

[9] 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`.

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

[11] "Repository of this project." (2024), [Online]. Available: `https://github.com/riesentoaster/klee-coreutils-experiments` (visited on Jan. 24, 2024).

[12] 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`.

[13] "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

```
/* Retrieve information about a FILE stream.
   Copyright (C) 2007 Free Software Foundation, Inc.

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 3 of the License, or
   (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program.  If not, see <http://www.gnu.org/licenses/>.  */

#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; \
```

```c
                       } *) 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
}
```

2 Spread Plots of All Measurements

Plots for individual utils are available in the project repository [11].

2.1 By Timeout

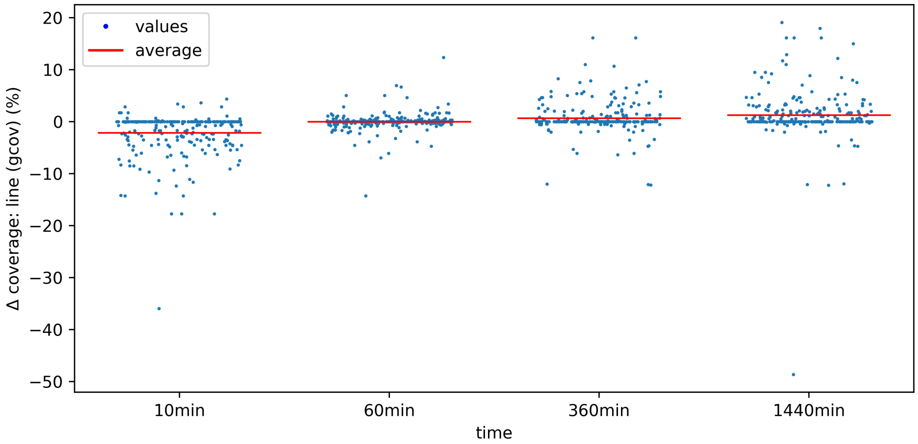

Figure 18: Spread of coverage as measured by gcov by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

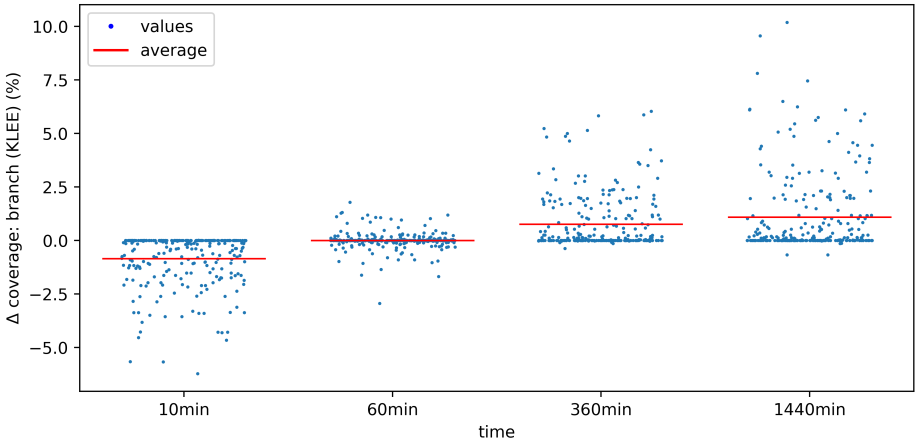

Figure 19: Spread of branch coverage as measured by KLEE by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

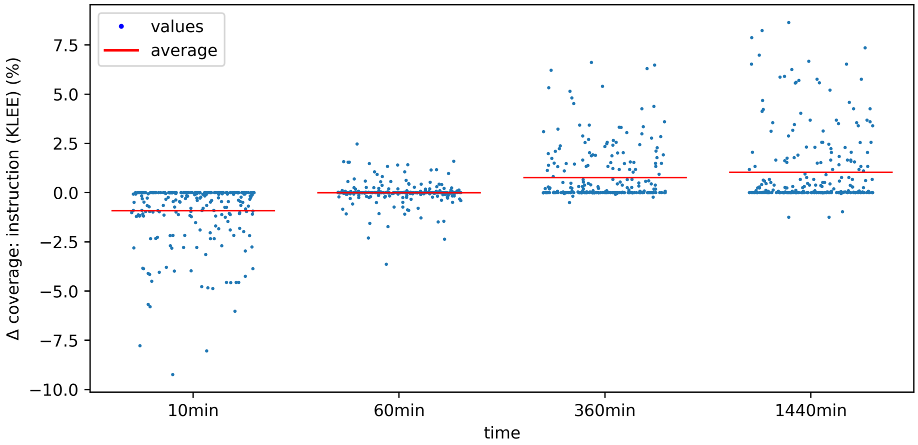

FIGURE 20: Spread of instruction coverage as measured by KLEE by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

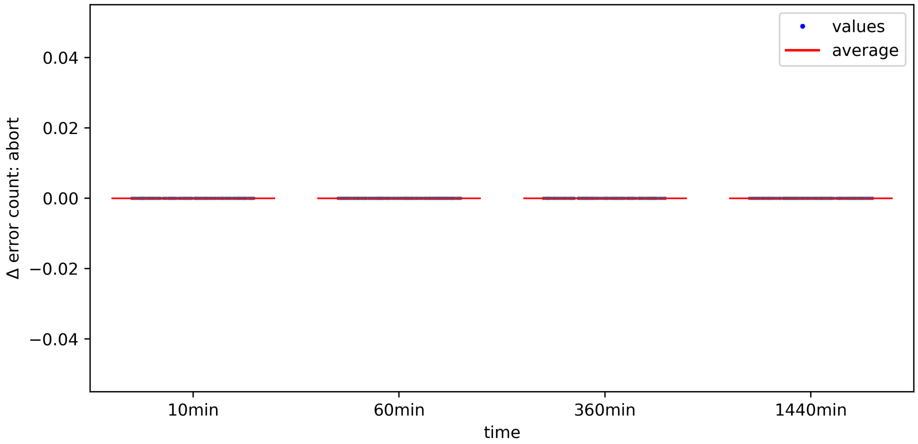

FIGURE 21: Spread of number of errors of type abort by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

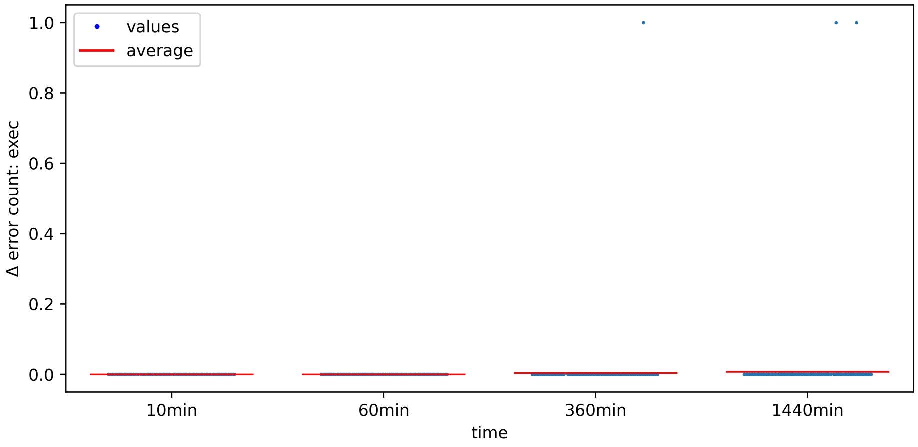

Figure 22: Spread of number of errors of type exec by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

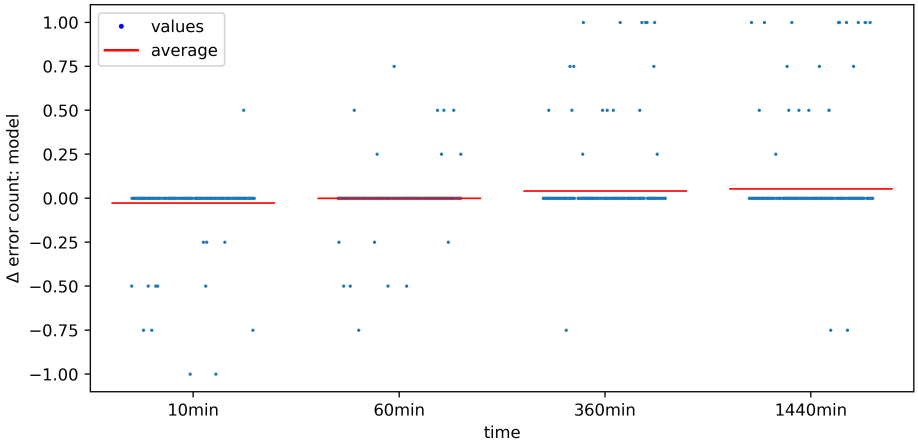

Figure 23: Spread of number of errors of type model by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

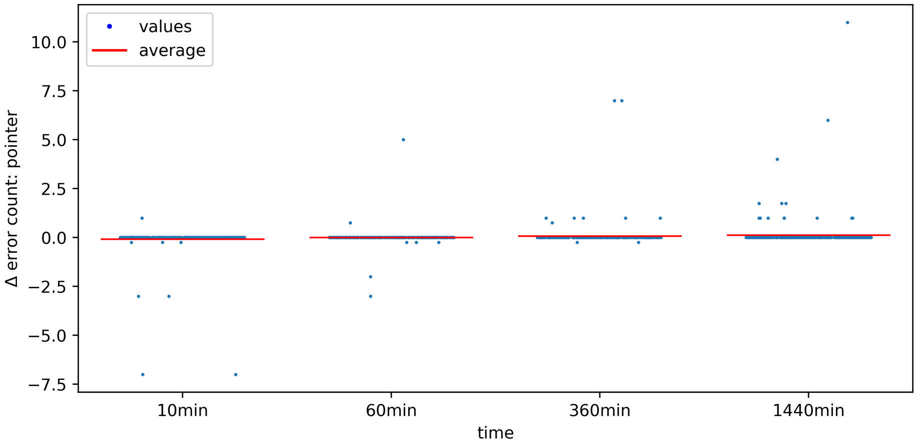

FIGURE 24: Spread of number of errors of type ptr by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

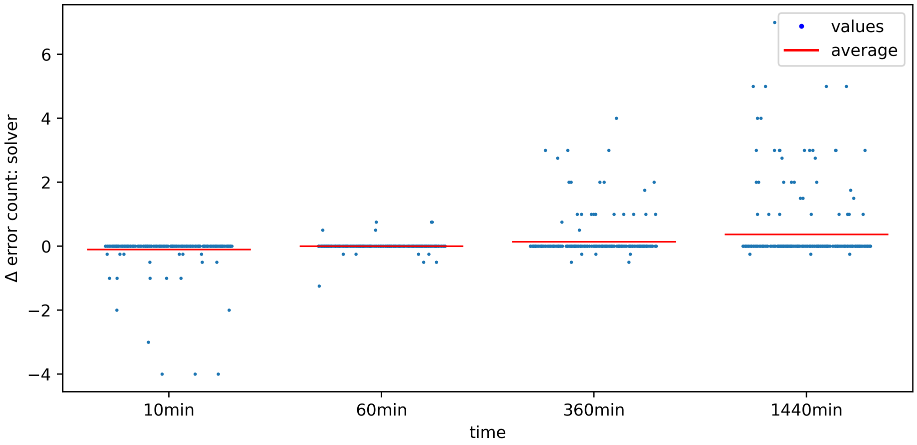

FIGURE 25: Spread of number of errors of type solver by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

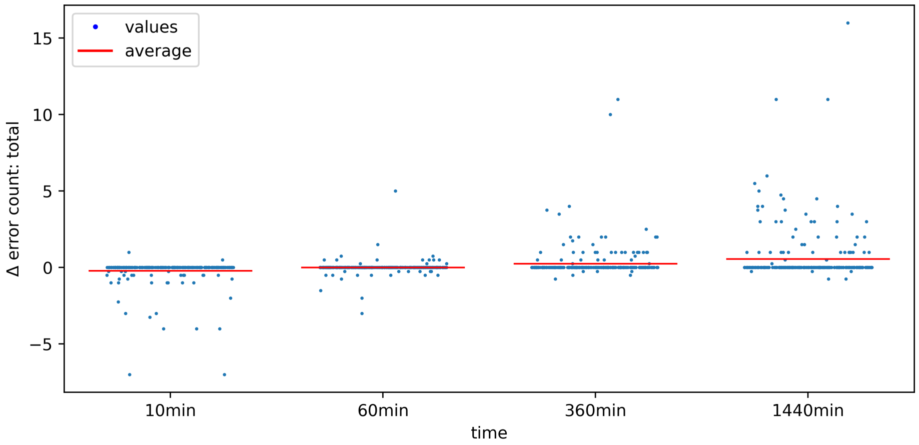

Figure 26: Spread of total number of errors by run, normalized to the mean of the util at a timeout of 60 minutes, across different timeouts

2.2 By Version

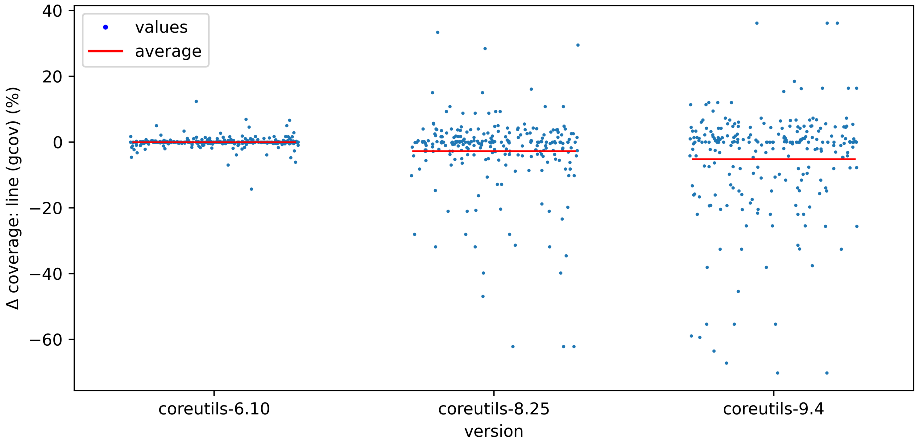

FIGURE 27: Spread of coverage as measured by `gcov` by run, normalized to the mean of the util at version 6.10, across different versions

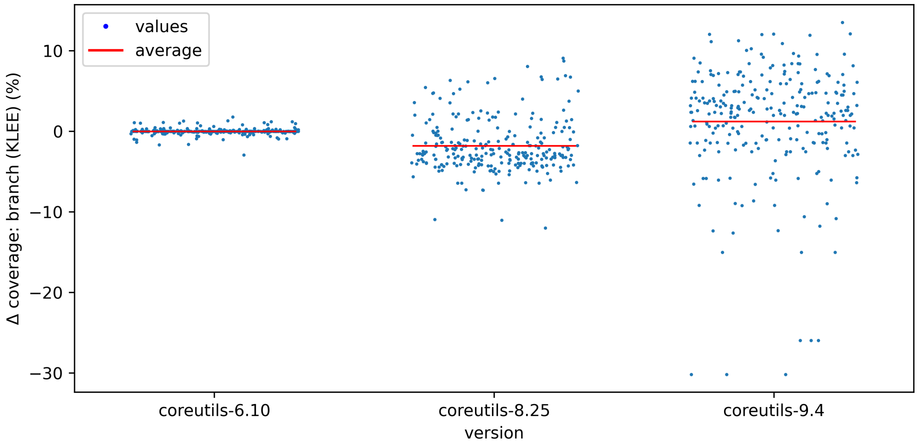

FIGURE 28: Spread of branch coverage as measured by KLEE by run, normalized to the mean of the util at version 6.10, across different versions

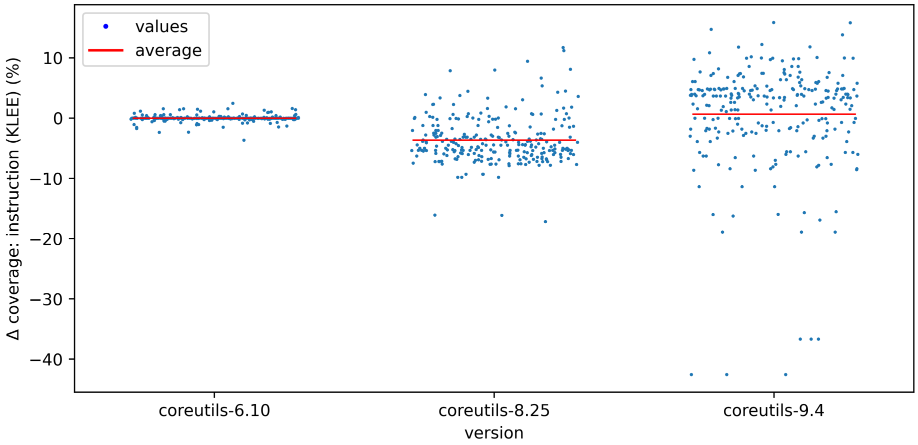

Figure 29: Spread of instruction coverage as measured by KLEE by run, normalized to the mean of the util at version 6.10, across different versions

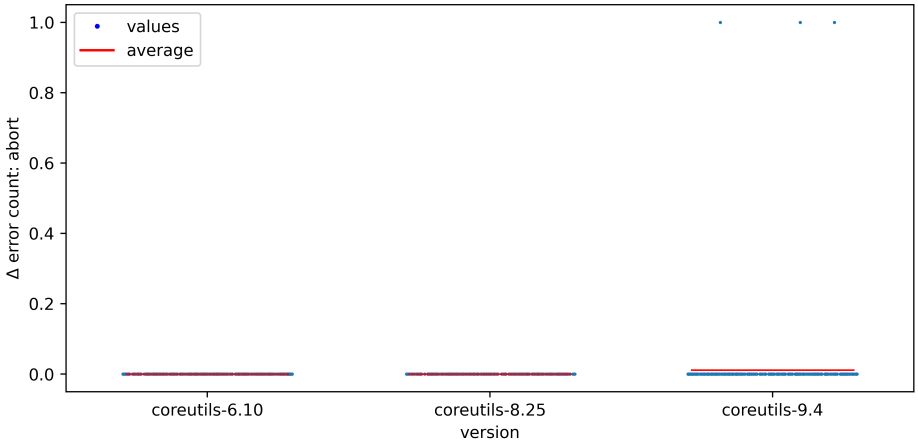

Figure 30: Spread of number of errors of type abort by run, normalized to the mean of the util at version 6.10, across different versions

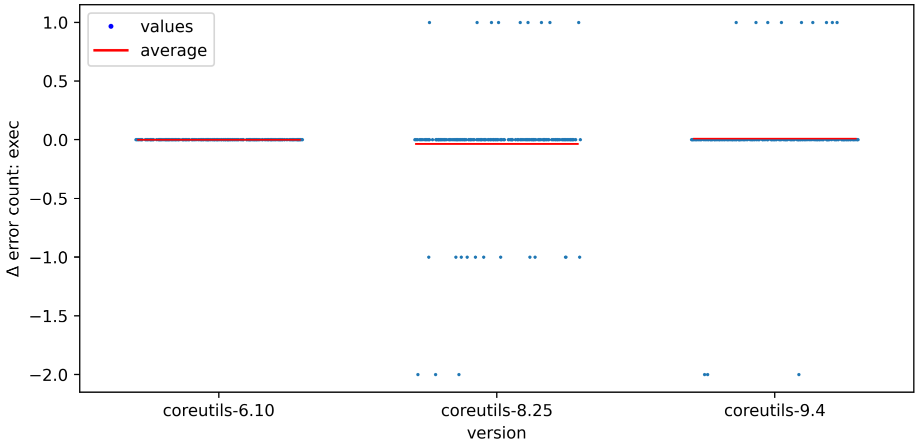

Figure 31: Spread of number of errors of type exec by run, normalized to the mean of the util at version 6.10, across different versions

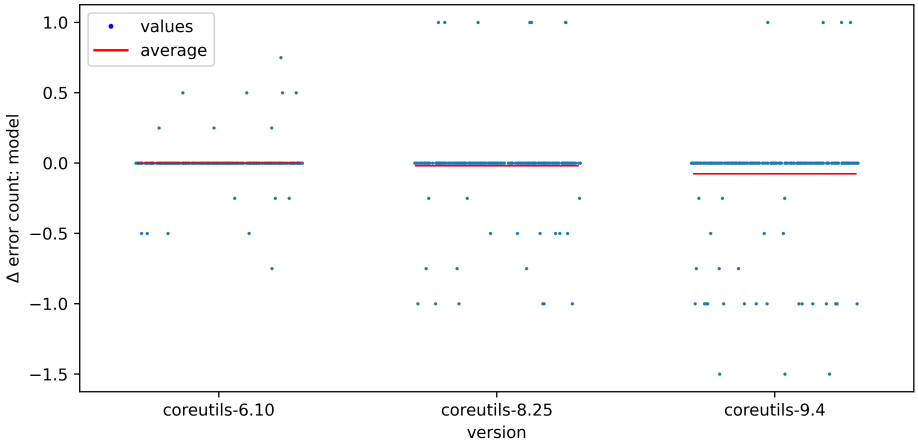

Figure 32: Spread of number of errors of type model by run, normalized to the mean of the util at version 6.10, across different versions

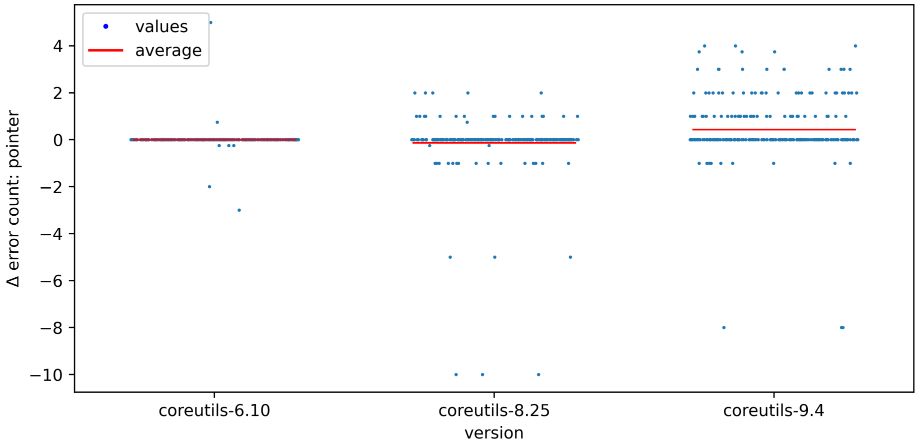

Figure 33: Spread of number of errors of type ptr by run, normalized to the mean of the util at version 6.10, across different versions

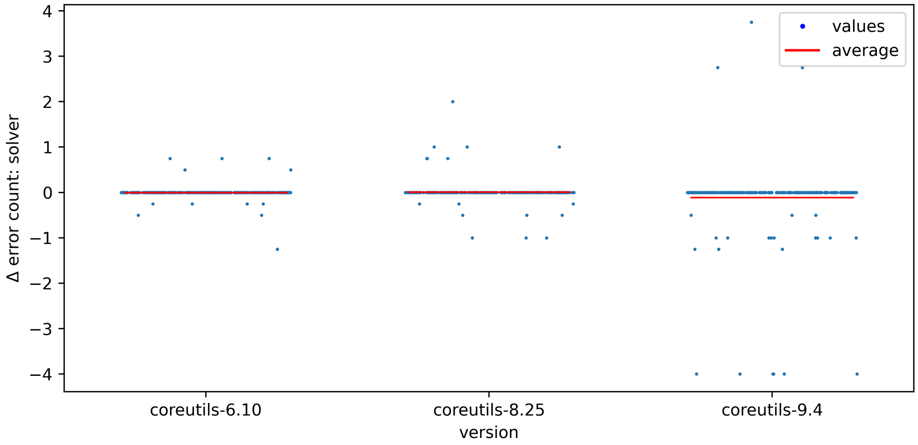

Figure 34: Spread of number of errors of type solver by run, normalized to the mean of the util at version 6.10, across different versions

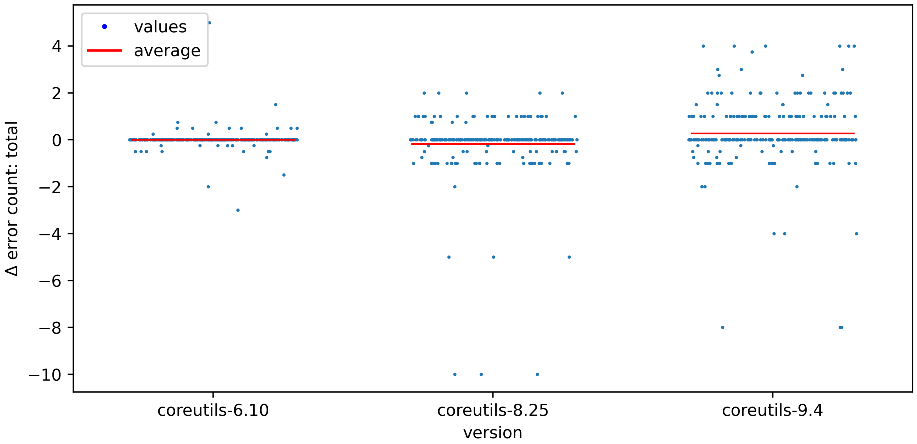

Figure 35: Spread of total number of errors by run, normalized to the mean of the util at version 6.10, across different versions