Running KLEE on GNU coreutils

Valentin Huber

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

February 10, 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?
   (c) How do the measured metrics compare to what was published 15 years ago?
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 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. It further makes complex build steps reproducible and acts as documentation.

However, the Docker image provided at this time is based on Ubuntu 22 (Jammy), and no longer builds coreutils 6.10 with the GNU Compiler (GCC). This is because its build system checks 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.

3.1.1 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.

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
```

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 '{}'
```

3.1.2 LLVM

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) [8], 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 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.

3.1.3 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

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
```

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 3.1.1. 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 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.

3.1.4 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 3.1.2 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 3.1.5. 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.

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}"
```

3.1.5 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.

3.1.6 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.

3.2 Analyzing a Single Run

3.2.1 Gathered Metrics

3.2.2 Comparison to the Original Paper

4 Comparing Runs

4.1 Metrics Distribution

4.2 Influence of Testing Timeout

5 Testing More Recent Versions of coreutils

5.1 Differences in Testing Setup

5.2 Findings

6 Discussion

6.1 Research Questions

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

Appendix

1 Error When Attempting to Build 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.

```
17 #include <config.h>
18 
19 /* Specification.  */
20 #include "freadahead.h"
21 
22 size_t
23 freadahead (FILE *fp)
24 {
25 #if defined _IO_ferror_unlocked     /* GNU libc, BeOS */
26   if (fp->_IO_write_ptr > fp->_IO_write_base)
27     return 0;
28   return fp->_IO_read_end - fp->_IO_read_ptr;
29 #elif defined __sferror             /* FreeBSD, NetBSD, OpenBSD, MacOS X, Cygwin */
30   if ((fp->_flags & __SWR) != 0 || fp->_r < 0)
31     return 0;
32   return fp->_r;
33 #elif defined _IOERR                /* AIX, HP-UX, IRIX, OSF/1, Solaris, mingw */
34 # if defined __sun && defined _LP64 /* Solaris/{SPARC,AMD64} 64-bit */
35 #  define fp_ ((struct { unsigned char *_ptr; \
36                          unsigned char *_base; \
37                          unsigned char *_end; \
38                          long _cnt; \
39                          int _file; \
40                          unsigned int _flag; \
41                        } *) fp)
42   if ((fp_->_flag & _IOWRT) != 0)
43     return 0;
44   return fp_->_cnt;
45 # else
46   if ((fp->_flag & _IOWRT) != 0)
47     return 0;
48   return fp->_cnt;
49 # endif
50 #elif defined __UCLIBC__            /* uClibc */
51 # ifdef __STDIO_BUFFERS
52   if (fp->__modeflags & __FLAG_WRITING)
53     return 0;
54   return fp->__bufread - fp->__bufpos;
55 # else
56   return 0;
57 # endif
58 #elif defined __QNX__               /* QNX */
59   if ((fp->_Mode & 0x2000 /* _MWRITE */) != 0)
60     return 0;
61   /* fp->_Buf <= fp->_Next <= fp->_Rend */
62   return fp->_Rend - fp->_Next;
63 #else
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."
65 #endif
66 }
```