master branch

The 2nd International KLEE Workshop on Symbolic Execution is coming!

Join us from 14-15 September 2020 in London.

#TESTING COREUTILS

Tutorial on How to Use KLEE to Test GNU Coreutils

As a more detailed explanation of using KLEE, we will look at how we did our testing of GNU Coreutils using KLEE. Please follow the instructions in our OSDI'08 coreutils experiment description to reproduce the experiment setup from the paper. This tutorial assumes that you have configured and built KLEE with `uclibc` and `POSIX` runtime support. All tests were done on a 64-bit Linux machine.

Step 1: Build coreutils with gcov

First you will need to download and unpack the source for coreutils. In this example we use version 6.11 (one version later than what was used for our OSDI paper) but you can use any version of Coreutils. However, for recent versions the `make -C src arch hostname` step can be skipped.

Before we build with LLVM, let's build a version of ***coreutils*** with ***gcov*** support. We will use this later

≡ KLEE

master branch

From inside the *coreutils* directory, we'll do the usual configure/make steps inside a subdire (obj-gcov). Here are the steps:

```
coreutils-6.11$ mkdir obj-gcov
coreutils-6.11$ cd obj-gcov
obj-gcov$ ../configure --disable-nls CFLAGS="-g -fprofile-arcs -ft
... verify that configure worked ...
obj-gcov$ make
obj-gcov$ make -C src arch hostname
... verify that make worked ...
```

We build with `--disable-nls` because this adds a lot of extra initialization in the C library which we are not interested in testing. Even though these aren't the executables that KLEE will be running on, we want to use the same compiler flags so that the test cases KLEE generates are most likely to work correctly when run on the uninstrumented binaries.

You should now have a set of `coreutils` in the `objc-gcov/src` directory. For example:

```
obj-gcov$ cd src
src$ ls -l ls echo cat
-rwxrwxr-x 1 klee klee 150632 Nov 21 21:58 cat
-rwxrwxr-x 1 klee klee 135984 Nov 21 21:58 echo
-rwxrwxr-x 1 klee klee 390552 Nov 21 21:58 ls
src$ ./cat --version
cat (GNU coreutils) 6.11
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

≡ KLEE

master branch

In addition, these executables should be built with `gcov` support, so if you run one it will write a `.gcda` into the current directory. That file contains information about exactly what code was executed when the program ran. See the [Gcov Documentation]() for more information. We can use the `gcov` tool itself to produce a human readable form of the coverage information. For example:

```
src$ rm -f *.gcda # Get rid of any stale gcov files
src$ ./echo**

src$ ls -l echo.gcda
-rw-rw-r-- 1 klee klee 896 Nov 21 22:00 echo.gcda
src$ gcov echo
File '../../src/echo.c'
Lines executed:24.27% of 103
Creating 'echo.c.gcov'

File '../../src/system.h'
Lines executed:0.00% of 3
Creating 'system.h.gcov'
```

By default `gcov` will show the number of lines executed in the program (the `.h` files include code which was compiled into `echo.c`).

Step 2: Install WLLVM

One of the difficult parts of testing real software using KLEE is that it must be first compiled so that the final program is an LLVM bitcode file and not a native binary. The software's build system may be set up to use tools such as 'ar', 'libtool', and 'ld', which do not in general understand LLVM bitcode files.

≡ KLEE master branch

program LLVM bitcode files from an unmodified C or C++ source package. WLLVM includes fo python executables: `wllvm` a C compiler and `wllvm++` a C++ compiler, the tool `extract-bc` extracting the bitcode from a build product (either object file, executable, library, or archive), and the sanity checker `wllvm-sanity-checker` for detecting configuration oversights. In this tutorial, we use WLLVM version 1.0.17.

To install *whole-program-llvm* the easiest way is to use `pip`:

```
$ pip install --upgrade wllvm
```

To successfully execute WLLVM it is necessary to set the environment variable `LLVM_COMPILER` to the underlying LLVM compiler (either `dragonegg` or `clang`). In this tutorial, we use `clang`:

```
$ export LLVM_COMPILER=clang
```

To make the environment variable persistent, add the export to your shell profile (e.g. `.bashrc`).

Step 3: Build Coreutils with LLVM

As before, we will build in a separate directory so we can easily access both the native executables and the LLVM versions. Here are the steps:

```
coreutils-6.11$ mkdir obj-llvm
coreutils-6.11$ cd obj-llvm
obj-llvm$ CC=wllvm ../configure --disable-nls CFLAGS="-g -O1 -Xcla
... verify that configure worked ...
obj-llvm$ make
obj-llvm$ make -C src arch hostname
... verify that make worked ...
```

KLEE

master branch

are going to test using KLEE, so we left of the `-fprofile-arcs -ftest-coverage` fl... Second, we added the `-O1 -Xclang -disable-llvm-passes` flags to `CFLAGS`. This is similar to adding `-O0`, however in LLVM 5.0 and later compiling with `-O0` prevents KLEE from performing its own optimisations (which we will do later). Therefore, we compile with `-O1` but explicitly disable all optimisations. See this issue for more details.

Note that we could have used `-O0 -Xclang -disable-O0-optnone` as well but because we are going to run Coreutils with optimisations later, it is better to compile with `-O1 -Xclang -disable-llvm-passes`. The `-O1` version emits bitcode that is more suited for optimisation, so we prefer to use that in this case.

`-D__NO_STRING_INLINES -D_FORTIFY_SOURCE=0 -U__OPTIMIZE__` is another set of **important flags**. In later versions of LLVM, `clang` emits safe version of certain library functions. For example it replaces `fprintf` with `__fprintf_chk`, which KLEE does not model. That means it will treat it as an external function and concretize state. It will lead to *unexpected results*.

If all went well, you should now have Coreutils *executables*. For example:

```
obj-llvm$ cd src
src$ ls -l ls echo cat
-rwxrwxr-x 1 klee klee 105448 Nov 21 12:03 cat
-rwxrwxr-x 1 klee klee  95424 Nov 21 12:03 echo
-rwxrwxr-x 1 klee klee 289624 Nov 21 12:03 ls
src$ ./cat --version
cat (GNU coreutils) 6.11
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Torbjorn Granlund and Richard M. Stallman.
```

master branch

WLLVM works in two steps. In the first step, WLLVM invokes the standard compiler and then, . object file, it also invokes a bitcode compiler to produce LLVM bitcode. WLLVM stores the locatio. the generated bitcode files in a dedicated section of the object file. When object files are linked together, the locations are concatenated to save the locations of all constituent files. After the build completes, one can use the WLLVM utility `extract-bc` to read the contents of the dedicated section and link all of the bitcode into a single whole-program bitcode file.

To obtain the LLVM bitcode version of all Coreutils, we can invoke `extract-bc` on all executable files:

```
src$ find . -executable -type f | xargs -I '{}' extract-bc '{}'
src$ ls -l ls.bc
-rw-rw-r-- 1 klee klee 543052 Nov 21 12:03 ls.bc
```

Step 4: Using KLEE as an interpreter

At its core, KLEE is just an interpreter for LLVM bitcode. For example, here is how to run the same `cat` command we did before, using KLEE. Note, this step requires that you configured and built KLEE with `uclibc` and `POSIX` runtime support (if you didn't, you'll need to go do that now).

```
src$ klee --libc=uclibc --posix-runtime ./cat.bc --version
KLEE: NOTE: Using klee-uclibc : /usr/local/lib/klee/runtime/klee-u
KLEE: NOTE: Using model: /usr/local/lib/klee/runtime/libkleeRuntim
KLEE: output directory is "/home/klee/coreutils-6.11/obj-llvm/src/
Using STP solver backend
KLEE: WARNING ONCE: function "vasnprintf" has inline asm
KLEE: WARNING: undefined reference to function: __ctype_b_loc
KLEE: WARNING: undefined reference to function: klee_posix_prefer_
KLEE: WARNING: executable has module level assembly (ignoring)
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 426374
```

```
KLEE: WARNING ONCE: calling external: vprintf(43649760, 514
cat (GNU coreutils) 6.11

License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by Torbjorn Granlund and Richard M. Stallman.
Copyright (C) 2008 Free Software Foundation, Inc.
KLEE: WARNING ONCE: calling close_stdout with extra arguments.

KLEE: done: total instructions = 28988
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

We got a lot more output this time! Let's step through it, starting with the KLEE command itself. The general form of a KLEE command line is first the arguments for KLEE itself, then the LLVM bitcode file to execute (cat.bc), and then any arguments to pass to the application (–version in this case, as before).

If we were running a normal native application, it would have been linked with the C library, but in this case KLEE is running the LLVM bitcode file directly. In order for KLEE to work effectively, it needs to have definitions for all the external functions the program may call. We have modified the uClibc C library implementation for use with KLEE; the –libc=uclibc KLEE argument tells KLEE to load that library and link it with the application before it starts execution.

Similarly, a native application would be running on top of an operating system that provides lower level facilities like write(), which the C library uses in its implementation. As before, KLEE needs definitions for these functions in order to fully understand the program. We provide a POSIX runtime which is designed to work with KLEE and the uClibc library to provide the majority of operating

≡ KLEE master branch

As before, cat prints out its version information (note that this time all the text is written out), but now have a number of additional information output by KLEE. In this case, most of these warnings are innocuous, but for completeness here is what they mean:

- *undefined reference to function: ___ctype_b_loc*: This warning means that the program contains a call to the function __ctype_b_loc, but that function isn't defined anywhere (we would have seen a lot more of these if we had not linked with uClibc and the POSIX runtime). If the program actually ends up making a call to this function while it is executing, KLEE won't be able to interpret it and may terminate the program.
- *executable has module level assembly (ignoring)*: Some file compiled in to the application had file level inline-assembly, which KLEE can't understand. In this case this comes from uClibc and is unused, so this is safe.
- *calling __user_main with extra arguments*: This indicates that the function was called with more arguments than it expected, it is almost always innocuous. In this case __user_main is actually the main() function for cat, which KLEE has renamed it when linking with uClibc. main() is being called with additional arguments by uClibc itself during startup, for example the environment pointer.
- *calling external: getpagesize()*: This is an example of KLEE calling a function which is used in the program but is never defined. What KLEE actually does in such cases is try to call the native version of the function, if it exists. This is sometimes safe, as long as that function does write to any of the programs memory or attempt to manipulate symbolic values. getpagesize(), for example, just returns a constant.

In general, KLEE will only emit a given warning once. The warnings are also logged to warnings.txt in the KLEE output directory.

Step 5: Introducing symbolic data to an application

We've seen that KLEE can interpret a program normally, but the real purpose of KLEE is to explore programs more exhaustively by making parts of their input symbolic. For example, lets look at running KLEE on the echo application.

≡ KLEE master branch

special function (klee_init_env) provided inside the runtime library. This function alters the , command line processing of the application, in particular to support construction of symbolic arguments. For example, passing –help yields:

```
src$ klee --libc=uclibc --posix-runtime ./echo.bc --help
...

usage: (klee_init_env) [options] [program arguments]
  -sym-arg <N>              - Replace by a symbolic argument with
  -sym-args <MIN> <MAX> <N> - Replace by at least MIN arguments ar
                              MAX arguments, each with maximum ler
  -sym-files <NUM> <N>      - Make NUM symbolic files ('A', 'B', '
                              each with size N
  -sym-stdin <N>            - Make stdin symbolic with size N.
  -sym-stdout               - Make stdout symbolic.
  -max-fail <N>             - Allow up to N injected failures
  -fd-fail                  - Shortcut for '-max-fail 1'
...
```

As an example, lets run echo with a symbolic argument of 3 characters.

```
src$ klee --libc=uclibc --posix-runtime ./echo.bc --sym-arg 3
KLEE: NOTE: Using klee-uclibc : /usr/local/lib/klee/runtime/klee-u
KLEE: NOTE: Using model: /usr/local/lib/klee/runtime/libkleeRuntim
KLEE: output directory is "/home/klee/coreutils-6.11/obj-llvm/src/
Using STP solver backend
KLEE: WARNING ONCE: function "vasnprintf" has inline asm
KLEE: WARNING: undefined reference to function: __ctype_b_loc
KLEE: WARNING: undefined reference to function: klee_posix_prefer_
KLEE: WARNING: executable has module level assembly (ignoring)
```

≡ KLEE master branch

```
..
KLEE: WARNING: calling close_stdout with extra arguments.
...
KLEE: WARNING ONCE: calling external: printf(42797984, 41639952)
..
KLEE: WARNING ONCE: calling external: vprintf(41640400, 52740448)
..
Echo the STRING(s) to standard output.

  -n             do not output the trailing newline
  -e             enable interpretation of backslash escapes
  -E             disable interpretation of backslash escapes (defa
      --help     display this help and exit
      --version  output version information and exit
Usage: ./echo.bc [OPTION]... [STRING]...
echo (GNU coreutils) 6.11
Copyright (C) 2008 Free Software Foundation, Inc.
If -e is in effect, the following sequences are recognized:

  \0NNN   the character whose ASCII code is NNN (octal)
  \\     backslash
  \a     alert (BEL)
  \b     backspace

License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

  \c     suppress trailing newline
  \f     form feed
  \n     new line
```

≡ KLEE master branch

```
  \v     vertical tab

NOTE: your shell may have its own version of echo, which usually 
the version described here.  Please refer to your shell's document
for details about the options it supports.

Report bugs to <bug-coreutils@gnu.org>.
Written by FIXME unknown.

KLEE: done: total instructions = 64546
KLEE: done: completed paths = 25
KLEE: done: generated tests = 25
```

The results here are slightly more interesting, KLEE has explored 25 paths through the program. The output from all the paths is intermingled, but you can see that in addition to echoing various random characters, some blocks of text also were output. You may be suprised to learn that coreutils' echo takes some arguments, in this case the options `--v` (short for `--version`) and `--h` (short for `--help`) were explored. We can get a short summary of KLEE's internal statistics by running klee-stats on the output directory (remember, KLEE always makes a symlink called klee-last to the most recent output directory).

```
src$ klee-stats klee-last
-----------------------------------------------------------------
|  Path    |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolv
-----------------------------------------------------------------
|klee-last|   64546|     0.15|    22.07|    14.14|   19943|
-----------------------------------------------------------------
```

Here *ICov* is the percentage of LLVM instructions which were covered, and *BCov* is the percentage of instructions or branches in the bitcode files; that includes a lot of library code which may no be executable. We can help with that problem (and others) by passing the `--optimize` option KLEE. This will cause KLEE to run the LLVM optimization passes on the bitcode module before executing it; in particular they will remove any dead code. When working with non-trivial applications, it is almost always a good idea to use this flag. Here are the results from running again with `--optimze` enabled:

```
src$ klee --optimize --libc=uclibc --posix-runtime ./echo.bc --sym
...
KLEE: done: total instructions = 33991
KLEE: done: completed paths = 25
KLEE: done: generated tests = 25
src$ klee-stats klee-last
------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolv
------------------------------------------------------------------
|klee-last|   33991|     0.13|    30.16|    21.91|    8339|
------------------------------------------------------------------
```

This time the instruction coverage went up by about six percent, and you can see that KLEE also ran faster and executed less instructions. Most of the remaining code is still in library functions, just in places that the optimizers aren't smart enough to remove. We can verify this – and look for uncovered code inside `echo` – by using KCachegrind to visualize the results of a KLEE run.

Step 6: Visualizing KLEE's progress with `kcachegrind`

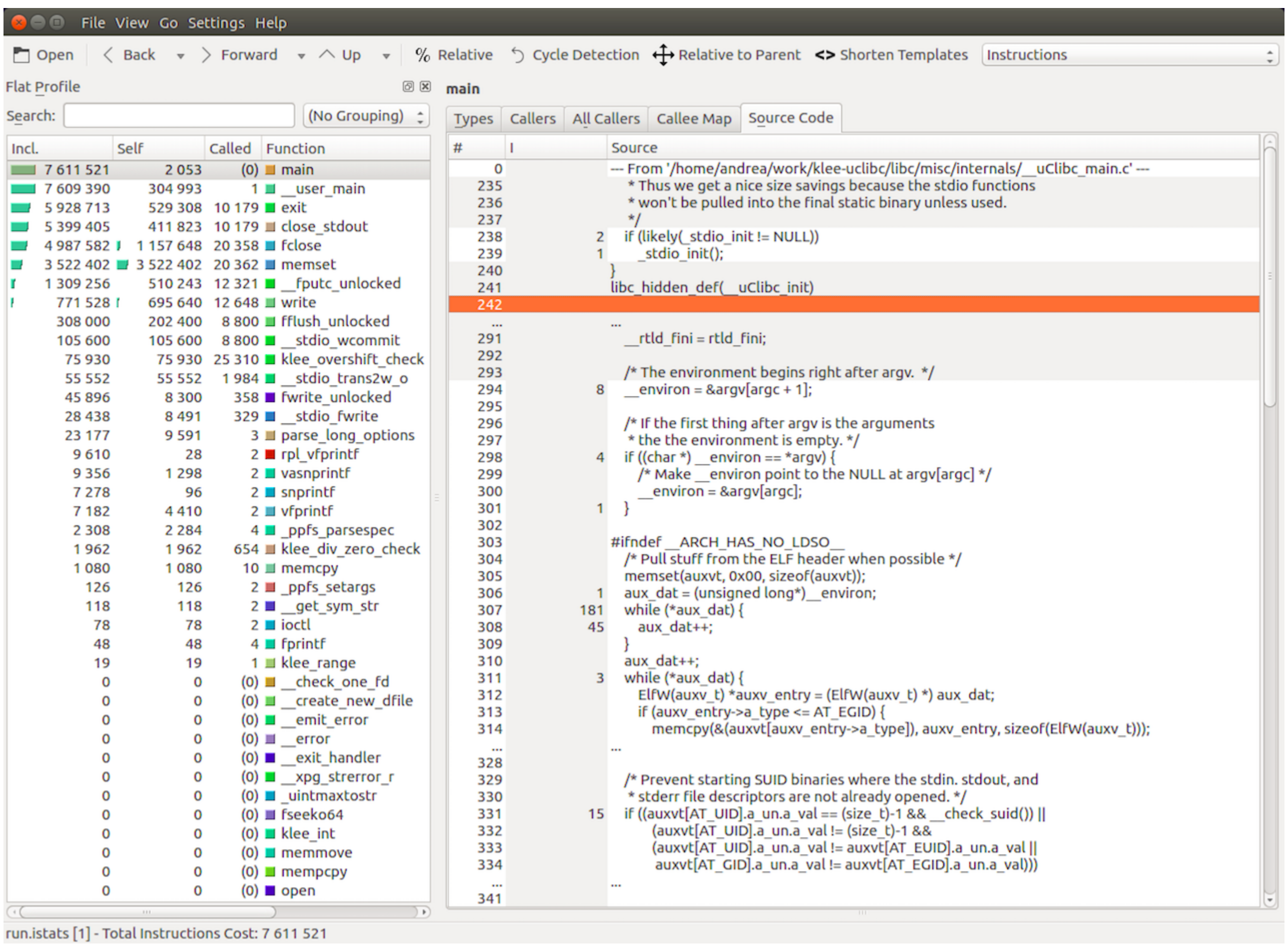
[KCachegrind](#) is an excellent profiling visualization tool, originally written for use with the callgrind plugin for valgrind. If you don't have it already, it is usually easily available on a modern Linux distribution via your platforms' software installation tool (e.g., `apt-get` or `yum`).

master branch

KCachegrind file. In this example, the `run.istats` is from a run without `--optimize`, so the results are easier to understand. Assuming you have KCachegrind installed, just run:

```
src$ kcachegrind klee-last/run.istats
```

After KCachegrind opens, you should see a window that looks something like the one below. You should make sure that the “Instructions” statistic is selected by choosing “View” > “Primary Event Type” > “Instructions” from the menu, and make sure the “Source Code” view is selected (the right hand pane in the screenshot below).

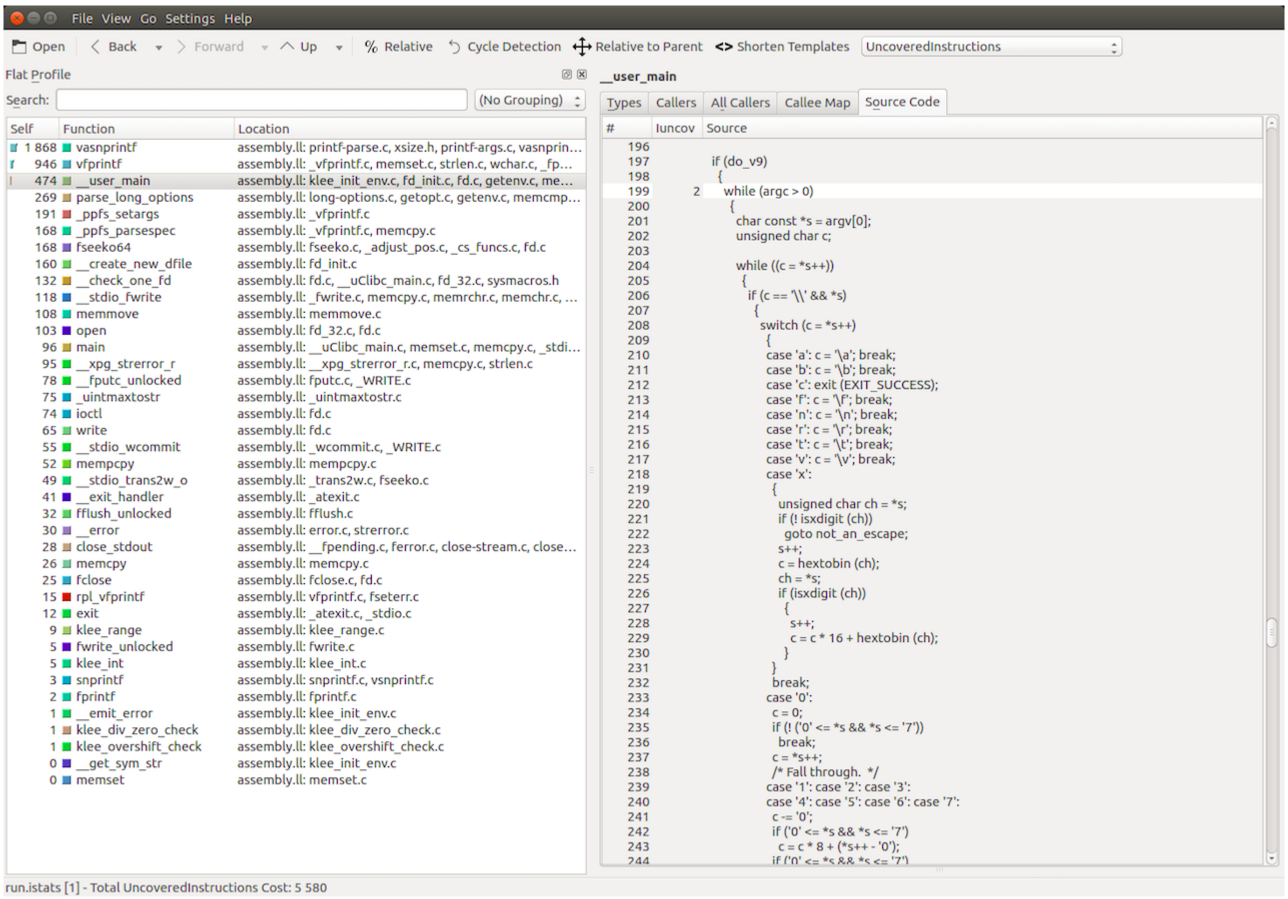


KCachegrind is a complex application in itself, and interested users should see the KCachegrind

master branch

"Self" column is the number of instructions which were executed in the function itself, and the (inclusive) column is the number of instructions which were executed in the function, or any of the functions it called (or its callees called, and so on).

KLEE includes quite a few statistics about execution. The one we are interested in now is "Uncovered Instructions", which will show which functions have instructions which were never executed. If you select that statistic and resort the list of functions, you should see something like this:



Notice that most of the uncovered instructions are in library code as we would expect. However, if we select the `__user_main` function, we can look for code inside `echo` itself that was uncovered. In this case, most of the uncovered instructions are inside a large `if` statement guarded by the variable `do_v9`. If you look a bit more, you can see that this is a flag set to true when `-e` is passed. The reason that KLEE never explored this code is because we only passed one symbolic argument –

One subtle thing to understand if you are trying to actually make sense of the KCachegrind r is that they include events accumulated across all states. For example, consider the following co

```
Line 1:      a = 1;
Line 2:      if (...)
Line 3:        printf("hello\n");
Line 4:      b = c;
```

In a normal application, if the statement on Line 1 was only executed once, then the statement on Line 4 could be (at most) executed once. When KLEE is running an application, however, it could fork and generate separate processes at Line 2. In that case, Line 4 may be executed more times than Line 1!

Another useful tidbit: KLEE actually writes the `run.istats` file periodically as the application is running. This provides one way to monitor the status of long running applications (another way is to use the klee-stats tool).

Step 7: Replaying KLEE generated test cases

Let's step away from KLEE for a bit and look at just the test cases KLEE generated. If we look inside the `klee-last` we should see 25 `.ktest` files.

```
src$ ls klee-last
assembly.ll       test000004.ktest  test000012.ktest  test000020.k
info              test000005.ktest  test000013.ktest  test000021.k
messages.txt      test000006.ktest  test000014.ktest  test000022.k
run.istats        test000007.ktest  test000015.ktest  test000023.k
run.stats         test000008.ktest  test000016.ktest  test000024.k
test000001.ktest  test000009.ktest  test000017.ktest  test000025.k
test000002.ktest  test000010.ktest  test000018.ktest  warnings.txt
```

master branch

These files contain the actual values to use for the symbolic data in order to reproduce the path KLEE followed (either for obtaining code coverage, or for reproducing a bug). They also contain additional metadata generated by the POSIX runtime in order to track what the values correspond to and the version of the runtime. We can look at the individual contents of one file using `ktest-tool`:

```
$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['./echo.bc', '--sym-arg', '3']
num objects: 2
object    0: name: 'arg0'
object    0: size: 4
object    0: data: '\x00\x00\x00\x00'
object    1: name: 'model_version'
object    1: size: 4
object    1: data: '\x01\x00\x00\x00'
```

In this case, the test case indicates that values "\x00\x00\x00\x00" should be passed as the first argument. However, `.ktest` files generally aren't really meant to be looked at directly. For the POSIX runtime, we provide a tool `klee-replay` which can be used to read the `.ktest` file and invoke the native application, automatically passing it the data necessary to reproduce the path that KLEE followed.

To see how it works, go back to the directory where we built the native executables:

```
src$ cd ..
obj-llvm$ cd ..
coreutils-6.11$ cd obj-gcov
obj-gcov$ cd src
```

≡ KLEE

master branch

To use the `klee-replay` tool, we just tell it the executable to run and the `.ktest` file to use. The program arguments, input files, etc. will all be constructed from the data in the `.ktest` file.

```
src$ klee-replay ./echo ../../obj-llvm/src/klee-last/test000001.kt
klee-replay: TEST CASE: ../../obj-llvm/src/klee-last/test000001.kt
klee-replay: ARGS: "./echo" ""

klee-replay: EXIT STATUS: NORMAL (0 seconds)
```

The first two and last lines here come from the `klee-replay` tool itself. The first two lines list the test case being run, and the concrete values for arguments that are being passed to the application (notice this matches what we saw in the `.ktest` file earlier). The last line is the exit status of the program and the elapsed time to run.

We can also use the `klee-replay` tool to run a set of test cases at once, one after the other. Let's do this and compare the `gcov` coverage to the numbers we got from `klee-stats`:

```
src$ rm -f *.gcda # Get rid of any stale gcov files
src$ klee-replay ./echo ../../obj-llvm/src/klee-last/*.ktest
klee-replay: TEST CASE: ../../obj-llvm/src/klee-last/test000001.kt
klee-replay: ARGS: "./echo" "@@@"
@@@
klee-replay: EXIT STATUS: NORMAL (0 seconds)
_..._
klee-replay: TEST CASE: ../../obj-llvm/src/klee-last/test000022.kt
klee-replay: ARGS: "./echo" "--v"
echo (GNU coreutils) 6.11
Copyright (C) 2008 Free Software Foundation, Inc.
```

≡ KLEE

master branch

```
src$ gcov echo
File '../../src/echo.c'
Lines executed:52.43% of 103
Creating 'echo.c.gcov'

File '../../src/system.h'
Lines executed:100.00% of 3
Creating 'system.h.gcov'
```

The number for `echo.c` here significantly higher than the `klee-stats` number because `gcov` is only considering lines in that one file, not the entire application. As with `kcachegrind`, we can inspect the coverage file output by `gcov` to see exactly what lines were covered and which weren't. Here is a fragment from the output:

```
        -:  193:      }
        -:  194:
       23:  195:just_echo:
        -:  196:
       23:  197:  if (do_v9)
        -:  198:    {
       10:  199:      while (argc > 0)
        -:  200:        {
    #####:  201:          char const *s = argv[0];
        -:  202:          unsigned char c;
        -:  203:
    #####:  204:          while ((c = *s++))
        -:  205:            {
    #####:  206:              if (c == '\\' && *s)
        -:  207:                {
    #####:  208:                  switch (c = *s++)
```

≡ KLEE master branch

```
#####:  211:            case 'b': c = '\b'; brea
#####:  212:            case 'c': exit (EXIT_SUCCE
#####:  213:            case 'f': c = '\f'; break;
#####:  214:            case 'n': c = '\n'; break;
```

The far left hand column is the number of times each line was executed; `-` means the line has no executable code, and `#####` means the line was never covered. As you can see, the uncovered lines here correspond exactly to the uncovered lines as reported in `kcachegrind`.

Before moving on to testing more complex applications, lets make sure we can get decent coverage of the simple `echo.c`. The problem before was that we weren't making enough data symbolic, providing echo with two symbolic arguments should be plenty to cover the entire program. We can use the POSIX runtime `--sym-args` option to pass multiple options. Here are the steps, after switching back to the `obj-llvm/src` directory:

```
src$ klee --only-output-states-covering-new --optimize --libc=ucli
...
KLEE: done: total instructions = 7611521
KLEE: done: completed paths = 10179
KLEE: done: generated tests = 57
```

The format of the `--sym-args` option actually specifies a minimum and a maximum number of arguments to pass and the length to use for each argument. In this case `--sym-args 0 2 4` says to pass between 0 and 2 arguments (inclusive), each with a maximum length of four characters.

We also added the `--only-output-states-covering-new` option to the KLEE command line. By default KLEE will write out test cases for every path it explores. This becomes less useful once the program becomes larger, because many test cases will end up exercise the same paths, and computing (or even reexecuting) each one wastes time. Using this option tells KLEE to only

≡ KLEE

master branch

the code, it only needed to write 57 test cases.

If we go back to the `obj-gcov/src` directory and rerun the latest set of test cases, we finally have reasonable coverage of `echo.c`:

```
src$ rm -f *.gcda # Get rid of any stale gcov files
src$ klee-replay ./echo ../../obj-llvm/src/klee-last/*.ktest
klee-replay: TEST CASE: ../../obj-llvm/src/klee-last/test000001.kt
klee-replay: ARGS: "./echo"

...

src$ gcov echo
File '../../src/echo.c'
Lines executed:97.09% of 103
Creating 'echo.c.gcov'

File '../../src/system.h'
Lines executed:100.00% of 3
Creating 'system.h.gcov'
```

The reasons for not getting perfect 100% line coverage are left as an exercise to the reader.

Step 8: Using `zcov` to analyze coverage

For visualizing the coverage results, you might want to use the zcov tool.

master branch

Resources

Mailing List

Doxygen

GitHub

TravisCI

Documentation for KLEE master branch