master branch

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

Join us from 14-15 September 2020 in London.

#TUTORIAL TWO

Testing a Simple Regular Expression Library

This is an example of using KLEE to test a simple regular expression matching function. You can find the basic example in the source tree under `examples/regexp`.

`Regexp.c` contains a simple regular expression matching function, and the bare bones testing harness (in `main()`) needed to explore this code with KLEE. You can see a version of the source code here.

This example will show to build and run the example using KLEE, as well as how to interpret the output, and some additional KLEE features that can be used when writing a test driver by hand.

We'll start by showing how to build and run the example, and then explain how the test harness works in more detail.

Building the example

bitcode format.

From within the `examples/regexp` directory:

```
$ clang -I ../../include -emit-llvm -c -g -O0 -Xclang -disable-O0-
```

which should create a Regexp.bc file in LLVM bitcode format. The `-I` argument is used so that the compiler can find `klee/klee.h`, which contains definitions for the intrinsic functions used to interact with the KLEE virtual machine. `-c` is used because we only want to compile the code to an object file (not a native executable), and finally `-g` causes additional debug information to be stored in the object file, which KLEE will use to determine source line number information. `-O0 -Xclang -disable-O0-optnone` is used to compile without any optimisation, but without preventing KLEE from performing its own optimisations, which compiling with `-O0` would.

If you have the LLVM tools installed in your path, you can verify that this step worked by running llvm-nm on the generated file:

```
$ llvm-nm Regexp.bc
         t matchstar
         t matchhere
         T match
         T main
         U klee_make_symbolic_name
         d LC
         d LC1
```

Normally before running this program we would need to link it to create a native executable. However, KLEE runs directly on LLVM bitcode files – since this program only has a single file there is no need to link. For “real” programs with multiple inputs, the llvm-link tools can be used in place of

master branch

Executing the code with KLEE

The next step is to execute the code with KLEE:

```
$ klee --only-output-states-covering-new Regexp.bc
KLEE: output directory = "klee-out-1"
KLEE: ERROR: .../klee/examples/regexp/Regexp.c:23: memory error: c
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: .../klee/examples/regexp/Regexp.c:25: memory error: c
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 6334861
KLEE: done: completed paths = 7692
KLEE: done: generated tests = 22
```

On startup, KLEE prints the directory used to store output (in this case `klee-out-1`). By default klee will use the first free `klee-out-N` directory and also create a `klee-last` symlink which will point to the most recent created directory. You can specify a directory to use for outputs using the `-output-dir=path` command line argument.

While KLEE is running, it will print status messages for “important” events, for example when it finds an error in the program. In this case, KLEE detected two invalid memory accesses on lines 23 and 25 of our test program. We’ll look more at this in a moment.

Finally, when KLEE finishes execution it prints out a few statistics about the run. Here we see that KLEE executed a total of ~6 million instructions, explored 7,692 paths, and generated 22 test cases.

Note that many realistic programs have an infinite (or extremely large) number of paths through them, and it is common that KLEE will not terminate. By default KLEE will run until the user presses Control-C (i.e. klee gets a `SIGINT`), but there are additional options to limit KLEE’s runtime and

- `-max-time=seconds`: Halt execution after the given number of seconds.
- `-max-forks=N`: Stop forking after `N` symbolic branches, and run the remaining paths to termination.
- `-max-memory=N`: Try to limit memory consumption to `N` megabytes.

KLEE error reports

When KLEE detects an error in the program being executed it will generate a test case which exhibits the error, and write some additional information about the error into a file `testN.TYPE.err`, where `N` is the test case number, and `TYPE` identifies the kind of error. Some types of errors KLEE detects include:

- **ptr**: Stores or loads of invalid memory locations.
- **free**: Double or invalid `free()`.
- **abort**: The program called `abort()`.
- **assert**: An assertion failed.
- **div**: A division or modulus by zero was detected.
- **user**: There is a problem with the input (invalid klee intrinsic calls) or the way KLEE is being used.
- **exec**: There was a problem which prevented KLEE from executing the program; for example an unknown instruction, a call to an invalid function pointer, or inline assembly.
- **model**: KLEE was unable to keep full precision and is only exploring parts of the program state. For example, symbolic sizes to malloc are not currently supported, in such cases KLEE will concretize the argument.

KLEE will print a message to the console when it detects an error, in the test run above we can see that KLEE detected two memory errors. For all program errors, KLEE will write a simple backtrace into the `.err` file. This is what one of the errors above looks like:

```
Error: memory error: out of bound pointer
File: .../klee/examples/regexp/Regexp.c
```

```
    #0 00000146 in matchhere (re=14816471, text=14815301) at .
    #1 00000074 in matchstar (c, re=14816471, text=14815301) at .
    #2 00000172 in matchhere (re=14816469, text=14815301) at .../ku
    #3 00000074 in matchstar (c, re=14816469, text=14815301) at .../
    #4 00000172 in matchhere (re=14816467, text=14815301) at .../kle
    #5 00000074 in matchstar (c, re=14816467, text=14815301) at .../
    #6 00000172 in matchhere (re=14816465, text=14815301) at .../kle
    #7 00000231 in matchhere (re=14816464, text=14815300) at .../kle
    #8 00000280 in match (re=14816464, text=14815296) at .../klee/ex
    #9 00000327 in main () at .../klee/examples/regexp/Regexp.c:59
Info:
  address: 14816471
  next: object at 14816624 of size 4
  prev: object at 14816464 of size 7
```

Each line of the backtrace lists the frame number, the instruction line (this is the line number in the `assembly.ll` file found along with the run output), the function and arguments (including values for the concrete parameters), and the source information.

Particular error reports may also include additional information. For memory errors, KLEE will show the invalid address, and what objects are on the heap both before and after that address. In this case, we can see that the address happens to be exactly one byte past the end of the previous object.

Changing the test harness

The reason KLEE is finding memory errors in this program isn't because the regular expression functions have a bug, rather it indicates a problem in our test driver. The problem is that we are making the input regular expression buffer completely symbolic, but the match function expects it to be a null terminated string. Let's look at two ways we can fix this.

This makes our driver look like this:

```
int main() {
  // The input regular expression.
  char re[SIZE];

  // Make the input symbolic.
  klee_make_symbolic(re, sizeof re, "re");
  re[SIZE - 1] = '\0';

  // Try to match against a constant string "hello".
  match(re, "hello");

  return 0;
}
```

Making a buffer symbolic just initializes the contents to refer to symbolic variables, we are still free to modify the memory as we wish. If you recompile and run klee on this test program, the memory errors should now be gone.

Another way to accomplish the same effect is to use the `klee_assume` intrinsic function. `klee_assume` takes a single argument (an unsigned integer) which generally should some kind of conditional expression, and "assumes" that expression to be true on the current path (if that can never happen, i.e. the expression is provably false, KLEE will report an error).

We can use `klee_assume` to cause KLEE to only explore states where the string is null terminated by writing the driver like this:

```
int main() {
  // The input regular expression.
```

```
  // Make the input symbolic.
  klee_make_symbolic(re, sizeof re, "re");
  klee_assume(re[SIZE - 1] == '\0');

  // Try to match against a constant string "hello".
  match(re, "hello");

  return 0;
}
```

In this particular example, both solutions work fine, but in general `klee_assume` is more flexible:

- By explicitly declaring the constraint, this will force test cases to have the `'\0'` in them. In the first example where we write the terminating null explicitly, it doesn't matter what the last byte of the symbolic input is and KLEE is free to generate any value. In some cases where you want to inspect the test cases by hand, it is more convenient for the test case to show all the values that matter.
- `klee_assume` can be used to encode more complicated constraints. For example, we could use `klee_assume(re[0] != '^')` to cause KLEE to only explore states where the first byte is not `'^'`.

NOTE: One important caveat when using `klee_assume` with multiple conditions; remember that boolean conditionals like `'&&'` and `'||'` may be compiled into code which branches before computing the result of the expression. In such situations KLEE will branch the process *before* it reaches the call to `klee_assume`, which may result in exploring unnecessary additional states. For this reason it is good to use as simple expressions as possible to `klee_assume` (for example splitting a single call into multiple ones), and to use the `'&'` and `'|'` operators instead of the short-circuiting ones.

master branch

Resources

Mailing List

Doxygen

GitHub

TravisCI

Documentation for KLEE master branch