#TUTORIAL ONE

Testing a Small Function

This tutorial walks you through the main steps needed to test a simple function with KLEE. Here is our simple function:

```
int get_sign(int x) {
  if (x == 0)
    return 0;

  if (x < 0)
    return -1;
  else
    return 1;
}
```

You can find the entire code for this example in the source tree under `examples/get_sign`. A

master branch

Marking input as symbolic

In order to test this function with KLEE, we need to run it on *symbolic* input. To mark a variable as symbolic, we use the `klee_make_symbolic()` function (defined in `klee/klee.h`), which takes three arguments: the address of the variable (memory location) that we want to treat as symbolic, its size, and a name (which can be anything). Here is a simple `main()` function that marks a variable `a` as symbolic and uses it to call `get_sign()`:

```
int main() {
  int a;
  klee_make_symbolic(&a, sizeof(a), "a");
  return get_sign(a);
}
```

Compiling to LLVM bitcode

KLEE operates on LLVM bitcode. To run a program with KLEE, you first compile it to LLVM bitcode using `clang -emit-llvm`.

From within the `examples/get_sign` directory:

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

which should create a `get_sign.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, such as `klee_make_symbolic`. It is useful to build with `-g` to add debug information to the bitcode file, which we use to generate source line level statistics information.

optimisations for KLEE which can be enabled with KLEE's `--optimize` command line opt
However, in later version of LLVM versions (> 5.0) the `-O0` zero flag should NOT be used when compiling for KLEE as it prevents KLEE from doing its own optimisations. `-O0 -Xclang -disable-O0-optnone` should be used instead, see this issue for more details.

If you do not wish to replay the test cases as described later and don't care about debug information and optimisation, you can delete the `klee/klee.h` include and then compile `get_sign.c` with:

```
$ clang -emit-llvm -c get_sign.c
```

However, we recommend using the longer command above.

Running KLEE

To run KLEE on the bitcode file simply execute:

```
$ klee get_sign.bc
```

You should see the following output (assumes LLVM 3.4):

```
KLEE: output directory = "klee-out-0"

KLEE: done: total instructions = 31
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
```

There are three paths through our simple function, one where a is `0`, one where it is less than `0` and

directory (in our case `klee-out-0`) containing the test cases generated by KLEE. KLEE na
output directory `klee-out-N` where `N` is the lowest available number (so if we run KLEE agai
will create a directory called `klee-out-1`), and also generates a symbolic link called `klee-las`
to this directory for convenience:

```
$ ls klee-last/
assembly.ll     run.istats       test000002.ktest
info            run.stats        test000003.ktest
messages.txt    test000001.ktest warnings.txt
```

Please click here if you would like an overview of the files generated by KLEE. In this tutorial, we only focus on the actual test files generated by KLEE.

KLEE-generated test cases

Test cases generated by KLEE are written into files with `.ktest` extension. These are binary files which can be read with the *ktest-tool* utility. *ktest-tool* outputs different representations for the same object, for instance Python byte strings (data), integers (int) or ascii text (text). So let's examine each file:

```
$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
```

```
$ ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x01\x01\x01\x01'
object 0: hex : 0x01010101
object 0: int : 16843009
object 0: uint: 16843009
object 0: text: ....

$ ktest-tool klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x80'
object 0: hex : 0x00000080
object 0: int : -2147483648
object 0: uint: 2147483648
object 0: text: ....
```

In each test file, KLEE reports the arguments with which the program was invoked (in our case no arguments other than the program name itself), the number of symbolic objects on that path (only one in our case), the name of our symbolic object (`'a'`) and its size (`4`). The actual test itself is represented by the value of our input: `0` for the first test, `16843009` for the second and `-2147483648` for the last one. As expected, KLEE generated value `0`, one positive value (`16843009`), and one negative value (`-2147483648`). We can now run these values on a native

Replaying a test case

master branch

While we can run the test cases generated by KLEE on our program by hand, (or with the help of an existing test infrastructure), KLEE provides a convenient *replay library*, which simply replaces the call to `klee_make_symbolic` with a call to a function that assigns to our input the value stored in the `.ktest` file. To use it, simply link your program with the `libkleeRuntest` library and set the `KTEST_FILE` environment variable to point to the name of the desired test case:

```
$ export LD_LIBRARY_PATH=path-to-klee-build-dir/lib/:$LD_LIBRARY_P
$ gcc -I ../../include -L path-to-klee-build-dir/lib/ get_sign.c -
$ KTEST_FILE=klee-last/test000001.ktest ./a.out
$ echo $?
0
$ KTEST_FILE=klee-last/test000002.ktest ./a.out
$ echo $?
1
$ KTEST_FILE=klee-last/test000003.ktest ./a.out
$ echo $?
255
```

As expected, our program returns `0` when running the first test case, `1` when running the second one, and `255` (`-1` converted to a valid exit code value in the `0-255` range) when running the last one.

master branch

Resources

Mailing List

Doxygen

GitHub

TravisCI

Documentation for KLEE master branch