KLEE is an open-source symbolic execution tool, it uses a combination of ideas from both conventional (static) and dynamic symbolic execution. In this assignment, we will perform some simple exercises using KLEE.

Preliminaries:

1. Use the KLEE docker as per the instructions here: https://klee.github.io/docker/
2. You do not need to build KLEE, just need to use KLEE.
3. KLEE relevant binaries will be installed at klee_build/bin folder. Inside this folder, klee, which is the symbolic executor and the helper tools e.g., ktest-tool, which is used to translate KLEE generated test cases to human readable form.

Using KLEE for a simple example:

Consider the program shown below:

```
#include <stdio.h>
int main() {
      int x;
      klee_make_symbolic(&x, sizeof(x), "x");
      if (x < 5)
        printf("x is less than 5\n");
      else
          printf("x is greater than or equal to 5\n");
}
```

The call to klee_make_symbolic indicates the test inputs (for generation) to klee. Such a function takes three parameters: 1) the first parameter is the address of the input variable (&x), the second parameter is the size of the input variable (sizeof(x)) and the third parameter is a symbolic name for the input variable ("x").

Klee does not work on the source code, instead, it works on an intermediate representation called LLVM bitcode. Hence, the program first needs to be compiled into LLVM bitcode using clang as follows:

```
clang -emit-llvm -g -c test.c -o test.bc
```

Where test.c is your source file and test.bc will be produced as an output of compilation.

Next, run klee using the test.bc file as follows. This will perform a symbolic execution on the code:

```
klee --write-smt2s test.bc
```

The argument `--write-smt2s` instructs KLEE to print the symbolic path conditions for each explored path in SMT2 format.

If your symbolic execution works as expected, you will see the following output:

```
KLEE: output directory is "/home/klee/klee_build/test/mytest/klee-out-36"
KLEE: Using STP solver backend
KLEE: SAT solver: MiniSat
KLEE: WARNING: undefined reference to function: printf
KLEE: WARNING ONCE: calling external: printf(274889217280) at test.c:10 6
x is less than 5
x is greater than or equal to 5
KLEE: done: total instructions = 16
KLEE: done: completed paths = 2
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 2
```

You may ignore the warnings. Most importantly, you should see that there are two completed paths and two generated tests. Since symbolic execution aims to obtain path coverage, this is expected behavior, as the aforementioned program has exactly two paths.

Inspecting Test Cases generated by KLEE:

All test cases generated by KLEE are saved under the linked directory `klee-last` (see `test00000*.ktest` files), which will be created in the same directory you run KLEE. The test cases are saved in binary format. Hence, you need to use a tool called `ktest-tool` to decode them. As an example, decoding the first test is performed as follows:

```
ktest-tool test000001.ktest /* (to decode test000001.ktest) */
```

This should print an output as follows to show the human readable values of the test inputs:

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

Assignment 1 (20 marks):

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int test_loop(int a[]) {
        int y;
        int i = 0;
        while (i < 100) {
                if (a[i] < 10)
                        y++;
                else
                        y--;
                i++;
        }
        assert(y <= 100);
}
```

a. Assume you wish to perform concolic execution for the function `test_loop`. Write the test driver to facilitate the symbolic execution by KLEE. **(5 marks)**

b. Run the KLEE symbolic execution for around 15 seconds and force terminate the symbolic execution if it does not finish within 15 seconds. How many tests were generated by KLEE? Explain your observation. **(5 marks)**

c. Now modify the `test_loop` function such that its semantics remain unchanged and KLEE generates exactly one test for the modified function. Justify your answer. **(10 marks)**

Assignment 2 (25 marks):

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* cell structure, data type lengths are shown alongside */
typedef struct cell {
   long v; //8 bytes
   struct cell* next; //8 bytes
   struct cell* prev; //8 bytes
}cell;

void test_unit(cell* p, int x) {
   /* allocate one cell */
   p = (cell *) malloc (sizeof(cell));
   p->v = 0;
   p->next = p->prev = NULL;
   if (x > 0) {
     cell* cur = (cell *)((char *)p + sizeof(long));
     cur->next = cur->prev = p;
     if (p->prev != NULL)
         assert(0 && "You should get out of here!!!");  //buggy function
   }
}
```

a. Assume you wish to perform concolic execution only for input x and for the function test_unit. Write the test driver to facilitate the symbolic execution by KLEE. **(5 marks)**

b. Run the KLEE symbolic execution and explain why the resulting error message appears. **(5 marks)**

c. Now run the program (without klee related calls) in your host (i.e., not in the KLEE docker). Discuss whether you observe any difference as compared to the concolic execution using KLEE. If there is any difference, explain why such a difference might occur. **(5 marks)**

d. Now assume that you wish to instrument the program which checks for possible error, as thrown by KLEE (in step b), and gracefully terminates the program upon detecting such error. Modify the function test_unit with this objective and run KLEE to validate your change. **(10 marks)**

Submission Instruction:

Submit a single zip file named HW1.zip containing all your answers together with the code. For example, for Assignment 1 (a, c) as well as for Assignment 2 (a, d) you need to submit the *.c files.