Symbolic Execution Testing of TCP using KLEE-Float

Bala Subramanyam Duggirala
School of Computing
University of Nebraska-Lincoln
Lincoln NE USA
bduggirala2@huskers.unl.edu

Samarth V. Mutnal
School of Computing
University of Nebraska-Lincoln
Lincoln NE USA
smutnal2@huskers.unl.edu

Introduction

Symbolic execution engines are currently widely used to generate test cases for the code to make it robust and dependable. One such area where Symbolic Execution can be used is to test TCP protocol, which is the dominant internet protocol world-wide. However, to test TCP protocol thoroughly, we need to take into consideration the floating-point calculations that are implemented in the TCP code, especially in the algorithms that implement aspects like congestion control. The problem arises when the symbolic execution engine that is analyzing the code is not supported to run computations on floating-points. This might lead to bad paths being explored, and in turn result in ineffective coverage of test cases.

Since TCP protocol is widely used on today's internet, it is important that the algorithms that implement TCP are reliable, and this is possible only through rigorous testing before deployment. Code that is not robust or reliable will eventually lead to production failures that might be extremely costly to resolve, e.g., an outage at a remote/inaccessible location like the South Pole.

Thus, if tested thoroughly, covering even farthest edge cases using analysis techniques that involve extensions like floating point extension, the result would be a more reliable code. The aim of our project is the same. We want to pick up on the current research on analysis using symbolic execution, and explore the possibilities of using a symbolic execution engine's floating-point extension. We will use KLEE, and its floating-point extension, KLEE-Float for our analysis of snippets of TCP code. We run the same code both with and without FP support, and try to see how is the Floating-Point Extension better in finding any extra or different edge cases.

Background

In this section, we explain what is symbolic execution, how do symbolic execution tools automate the generation of test cases, floating point arithmetic and how it can cause problems in evaluation of a code, introduction to TCP, and how TCP uses floating-point arithmetic, a brief introduction to KLEE and how it works.

Symbolic Execution:

Symbolic execution is the process of exploring the code using symbolic inputs. Manually testing the code with every possible input can be a demanding task, so what symbolic execution does is, it tries to explore all the code using symbolic inputs. The symbolic inputs are used to, for example, execute either direction of an "if" statement, getting path conditions for both the scenarios where "if" is TRUE and when it is FALSE. KLEE is one such tool that implements symbolic execution to generate its test cases (refer [1]). Symbolic execution tools usually implement constraint solvers to automatically determine the input for particular test case.

Floating Points:

Floating points are usually approximations of the real numbers. They are mainly used to efficiently use storage space although a sacrifice of precision and range is demanded. The scientific notation of floating points is given by $(-1)^s \times m \times b^e$. Here s = 0 or 1, m is called the significand with range [0,b), b is base integer either 2 or 10 usually, e is exponent integer. For example, 3.25 can be written as 3.25 * 10^0.

It is well known that floating point evaluations are tricky when it comes to computer analysis, even though specific IEEE standard tools are involved. It is a big problem for constraint solving involving such floating points, where incorrect calculations over floating points lead to imperfect solutions, i.e., input test data. (refer [2])

TCP Protocol:

As already mentioned, one of the important protocols of internet protocol suite is TCP. TCP protocols are specified in five different layers, i.e., physical, link, network, transport and application layer.

Through years, TCP has evolved by implementing various algorithms like congestion control algorithms that make it much faster and more reliable. This algorithm especially uses variables and functions like snd_cwnd, reno, ack, etc., which are mostly based on calculations that involve floating points. So, it is important that such algorithms be analyzed using constraint solvers that are floating-point-supported.

KLEE:

KLEE is a symbolic execution tool that is used test scenarios with high coverage of complex codes. It is based on LLVM framework. KLEE works by converting the .c code into llvm bitcode (.bc file). The output is various test files, one for each test scenario, describing the input passed and other parameters.

Approach:

Our aim is to test simple functions involving floating points using KLEE both with and without floating point extension and compare them. We start by taking a basic code, that is present in the KLEE website's tutorial 1 and running KLEE on that. To test TCP code for floating point analysis, we pulled the "CUBIC" code from Dr. Lisong Xu 's (UNL) research (refer [6]) and implemented the same approach. Although we only tested this on a high-level, we chose this because floating-point extension of KLEE was not implemented while analyzing this code in the mentioned research.

In order to run KLEE, the easiest way is via a docker container. After installing docker on Ubuntu, we can clone KLEE into a container that will provide us with a KLEE runtime environment that can run test files.

Running KLEE-Float is slightly different, and a little more complicated, which we will explain in the README file. After creating respective directories, we run the test code using both versions of KLEE and try to compare the results.

The instructions to execute KLEE are provided in a separate instructions document(README).

In order to test any code, the input must be changed to a symbolic input using KLEE's "klee_make_symbolic()" function, which takes in a variable with a specific data type and converts into symbolic input to the program being tested. It is given as klee_make_symbolic(&a, sizeof(a), "a"). This is added as a part of a main() function which is used to return the function that is being tested using this symbolized input.

After adding the required main() function with symbolized inputs, the code is compiled to LLVM bit code for KLEE to run it. After compiling, a ".bc" file is generated and same is used to run KLEE.

After running KLEE on ."bc" file, a brief summary of how many paths were executed is shown and an output directory is created with test_output files. Each test file(ex: test000001.ktest) represents one test scenario which provides the details of what input was given, its name, size, etc., For our project, we tested two different codes to compare the results of KLEE and KLEE-Float.

Evaluation:

Test Scenario 1: We test the sample code from the KLEE website's tutorial1. We tested the same code using both KLEE and KLEE-Float and observed that although the test inputs were different in both the scenarios, no alternative paths were explored or new tests were generated. We believe this is because the code actually does not contain any floating-point calculations.

We can see the code tested in the following figure:

klee@00ecffc5c929: ~/SE_Project/tutorial1

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

            if (x < 0)
                      return -1;
              else
                        return 1;
}
int main() {
          int a;
            klee_make_symbolic(&a, sizeof(a), "a");
              return get_sign(a);
}
~
~
~
~
```

Test_Files generated using KLEE:

```
klee@00ecffc5c929:~/SE_Project/tutorial1$ vim test.c
klee@00ecffc5c929:~/SE_Project/tutorial1$ ktest-tool  klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['test.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
object 0: text: ....
klee@00ecffc5c929:~/SE_Project/tutorial1$ ktest-tool  klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['test.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: ....
klee@00ecffc5c929:~/SE_Project/tutorial1$ ktest-tool  klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['test.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: ....
klee@00ecffc5c929:~/SE_Project/tutorial1$
```

As seen from the above figure, three different test cases are generated with a positive, a negative and a 0 value as input to cover the three scenarios of the get_sign() code.

The same code, when run using KLEE-Float, gives the output:

```
est.c:11:2: warning: implicit declaration of function 'klee_make_symbolic' is invalid in C99 [-Wimplicit-function-declaration]
      klee_make_symbolic(&a, sizeof(a), "a");
      ^
 warning generated.
896144bc0fd:~ $ llvm-dis ./test.bc
896144bc0fd:~ $ klee test.bc
LEE: output directory is "/home/user/klee-out-0"
LEE: Using Z3 solver backend

LEE: done: total instructions = 30
LEE: done: completed paths = 3
LEE: done: generated tests = 3
896144bc0fd:~ $ ktest-tool  klee-last/test000001.ktest
test file : 'klee-last/test000001.ktest'
rgs       : ['test.bc']
um objects: 1
bject    0: name: b'a'
bject    0: size: 4
bject    0: data: b'\x00\x00\x00\x00'
896144bc0fd:~ $ ktest-tool  klee-last/test000002.ktest
test file : 'klee-last/test000002.ktest'
rgs       : ['test.bc']
um objects: 1
bject    0: name: b'a'
bject    0: size: 4
bject    0: data: b'\x00\x00\x00\x00 '
896144bc0fd:~ $ ktest-tool  klee-last/test000003.ktest
test file : 'klee-last/test000003.ktest'
rgs       : ['test.bc']
um objects: 1
bject    0: name: b'a'
bject    0: size: 4
bject    0: data: b'\xff\xff\xff\xff'
896144bc0fd:~ $
```

Although they are generated in a different format, after converting the input values mentioned in the above figure, when converted to int values, they are respectively 0, 32 and -1. Here, even though the code is explored using different inputs, no alternative paths have been explored by KLEE-float because there is no floating-point logic in the code to make a difference.

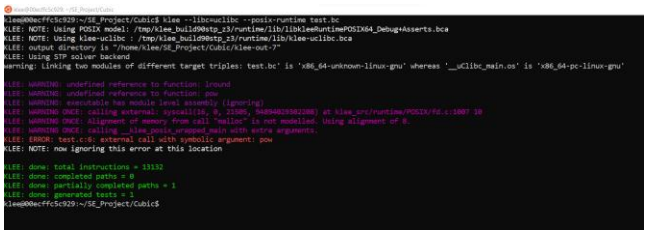
Test Scenario 2: Here, we extend our testing to a part of TCP avoidance algorithm "CUBIC". When we tried to analyze the "cubic" function of this algorithm, because of errors related to external function calls with symbolic arguments, we had to modify the code in order to reduce the number of such errors, assuming our testing wouldn't be affected. This code has 3 input variables using which a main function had been declared as follows:

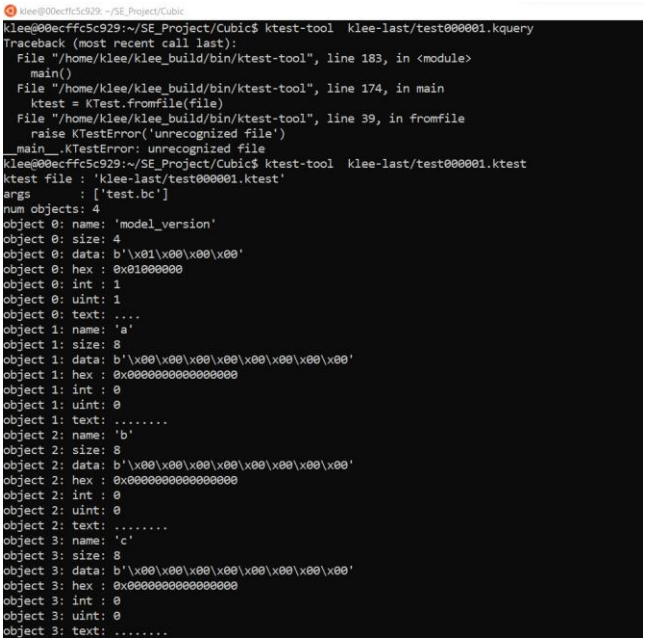
klee@00ecffc5c929: ~/SE_Project/Cubic

```
#include <math.h>
#include <stdio.h>
#include <sys/param.h>

static long cubic(double C, double RTT, double p) {
          double w = pow(RTT, 0.75);
            w = w / pow(p, 0.75);
              w = w * pow(C * 3.7 / 1.2, 0.25);
                const long cwnd = lround(w);

                  const long tcp_friend = p;

                    return 1;
}
int main() {
          double a;
          klee_make_symbolic(&a, sizeof(a), "a");
          double b;
          klee_make_symbolic(&b, sizeof(b), "b");
          double c;
          klee_make_symbolic(&c, sizeof(c), "c");
          return cubic(a, b, c);
}
~
~
```

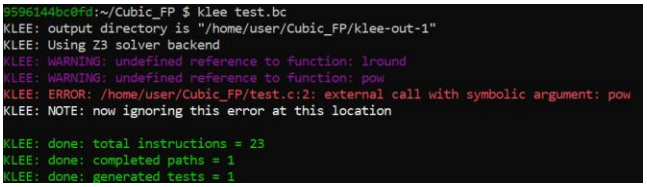
Interestingly with this code, the paths explored by KLEE and KLEE-float were not the same as can be seen below:

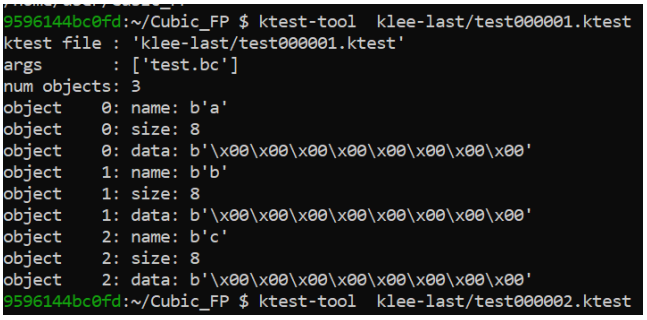
KLEE output:





KLEE-Float output:





We can see from the above results that KLEE-Float with FP extensions was able to completely explore an otherwise partially explored path without the FP extension of the same engine.

Conclusions and future work:

From our results, we could see that KLEE-Float has the potential to explore more accurate paths when floating-points are involved in the code. This shows that even when we would think a code might actually be reliable, we cannot be sure until we have tried every remote possibility we have..During our learning process, we came to know that a multitude of approaches are being implemented in symbolic testing, and various tools that implement various features are currently being used. For example,

QuickCheck tests universally quantified properties by testing random samples and "shrinking" the scope dynamically based on results of previous executions (refer [3]).

Symbolic PathFinder Tool, which has an interesting property where it not only supports floating point extensions but also has functions that checks the accuracy of the floating-point calculations (refer [4])

In our project, although we could not extensively test complex programs, based on our simple high-level analysis, we could see that KLEE with FP extension had produced different inputs for the same code compared to KLEE without FP. We believe that this area of testing has a lot of potential and we intend to explore multiple tools and compare them to possibly find ideas about enhancements of that can be implemented across these tools. For example, KLEE can have the same feature Symbolic Pathfinder has, i.e., an accuracy checking function to check the accuracy of floating-point computations.

Work Division:

Installing Docker – Bala, Installing Ubuntu, pulling KLEE – Samarth, Installing KLEE-Float—Bala, Running Test files and Analyzing output – Bala, Samarth, Analysis, test data gathering - Bala, Samarth, Test evidence documentation – Samarth, Introduction PPT– Bala, Samarth, Final Report –Bala, Samarth

REFERENCES

[1] KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Cristen Weise, Stefan Kowalewski, Klaus Wehrle.

[2] Floating-Point Symbolic Execution: A Case Study in N-Version Programming. Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F. Donaldson, Rafael Zahl, Klaus Wehrle.

[3] Automatic Testing of TCP/IP Implementations Using Quickcheck. Javier Paris, Thomas Arts.

[4] Symbolic Execution for Checking the Accuracy of Floating-Point Programs. Jaideep Ramachandran, Corina Pasareanu, Thomas Wahl.

[5] Automatic Detection of Floating-Point Exceptions. Earl T. Barr, Thanh Vo, Vu Le, Zhendong Su.

[6] Scalably Testing Congestion Contro; Algorithms of the Real-World TCP Implementations. Wei Sun, Lisong Xu, Sebastian Elbaum.

[7] Symbolic Crosschecking of Floating-Point and SIMD Code. Peter Collingbourne, Cristian Cadar, Paul H.J. Kelly.