Verification and Validation of Coreutils-2.18 using KLEE

EE382C: Verification and Validation of Software

Chris J Arges
University of Texas
Austin,Texas,USA
christopherarges [at] gmail [dot] com

Jennifer Kaser
University of Texas
Austin,Texas,USA
jennifer.kaser [at] gmail [dot] com

1. ABSTRACT

As software defects become an increasing problem and cost significant amounts of money when not fixed, there is a growing need to have efficient ways of detecting faults. In this paper, KLEE is used examine an open source project to evaluate the tool and its effectiveness in detecting software defects [2]. Additionally, the tool relevance and applicability to real world applications will be evaluated. Initial use of the tool has shown that it has not kept up with recent versions of tools that it utilizes. Also, there are some areas with respect to setup and execution that could be refined to make it easier to utilize as a development tool. Additionally, we were unable to produce comparable results of test coverage in the original KLEE paper of Coreutils with version 8.16. Although there are features of the tool that allow simulation of the environment, typical real world applications often rely on complex libraries which very quickly make computation using symbolic execution unrealistic for most applications. At this point in the tool's maturity, the best application seems to be self contained executables not dependant on a large number of libraries. While we did not find any new defects in our test runs, it was clear there is still much work to be done to use KLEE in large-scale programs.

2. INTRODUCTION

Purpose

Detecting defects in applications is critical to reduce the maintenance costs of software. The ability to exhaustively check a program for problems leading to critical failures is essential for high reliability applications. This paper examines KLEE as a tool to generate a high coverage test suite and to find defects. The goal of this paper is to evaluate the tool and its applicability in industry, identify any potential issues in its use, and try and find potential issues with a new revision of the Coreutils that KLEE was originally run with.

Symbolic Execution

Symbolic execution is a method of running a computer program by using symbols in place of actual values during execution. Instead of executing a program with concrete inputs, the program is executed symbolically with a set of classes of inputs [5]. Model checking or testing are two common techniques that can be used to ensure that programs behave correctly. Testing will only be as good as the test cases that are specified, and thus model checking provides a more exhaustive solution to determining test cases. However, it may be improbable to check all of these test cases due to the state explosion problem. Symbolic execution provides a way to reduce this state explosion problem by executing symbolically instead of exhaustively testing each value [4]. A symbolic execution tree is a good illustration of how one can show various code paths without actually assigning values or evaluating expressions. Figure 2 shows an example of a symbolic execution tree on a simple if/else statement in C. As we can see, we assign symbols of X and Y to the variables x and y and show the conditional assignments. Thus, if X > Y, x = X+Y is evaluated; otherwise X <= Y is true and x = X-Y.

`int x, y;`	`x=X, y=Y`
`if (x>y) {`	`X>?Y`
`x = x + y;`	`[X>Y] x = X +Y`
`} else {`	
`x = x - y;`	`[X<=Y] x = X-Y`
`}`	

Table 1: Example of a Symbolic Execution Tree

Overview of KLEE

After exploring several different options for tools and methods used to detect bugs in source code, we settled on using a symbolic execution tool called KLEE. This tool was chosen due to it's availability of source code, ability to analyze C code, success in finding defects, and ability to detect certain classes of defects in existing code without any modification. It is a tool which uses symbolic execution to exhaustively explore the state space of a program and generates inputs which could result in a crash. KLEE works by utilizing the LLVM compiler to generate byte code for an application. The user then uses KLEE to run the application with a defined number of symbolic arguments. This can either be done through a command line utility or compiled into the application itself. When the application is executed, KLEE uses the symbolic arguments and determines the different execution paths and the path conditions when executing the program. KLEE then executes each different execution path by processing the LLVM byte code and tracking additional state information. The state information is used to detect errors in memory accesses, multiple calls to free memory, calls to abort, assertion failures and division by zero. Once errors are detected, a utility is provided to generate a test case input of the symbolic arguments that results in that execution path with the error to be exercised. In the initial paper for KLEE, there was mention of systematically simulating errors to system calls [2].

Related Work

In a paper published in 2008, the KLEE tool was used to analyze source code from the coreutils project and successfully find defects that had been unreported for over 15 years very heavily used utilities [2]. Recently, there has been much discussion with utilizing symbolic execution for detecting bugs. As a result, a paper has been published last year which provides an overview of various tools as well as examples of uses in both academia and industry [3].

3. RUNNING KLEE

Project Environment Setup

The LLVM-KLEE environment requires certain versions of tools in order to work correctly. Building and installation of the tool went smoothly when using the same versions of tools which KLEE used when it was developed. Using newer versions caused several problems. For example, when running the 2.18 release of coreutils with KLEE, the newer version used a different compiler flag and an extension of C which was not compatible with llvm-gcc-4.2 used by KLEE and required some Make environment flags to be modified to be able to be compiled. In addition parts of coreutils-2.18 like libstdbuf and man pages could not be properly built because of errors and were thus removed from the build. Using llvm-gcc-4.6 which is included in Ubuntu 12.04 and most newer Linux distribution caused issues because of major differences in the LLVM project between 4.2 and 4.6. Thus, and older and supported version was used which better matched the build environment described on the KLEE website. For our tests, we ran KLEE on a virtual machine running the Ubuntu 10.04 Linux distribution. Initially and for the simple examples, this worked fine but very soon performance became an issue even when giving the VM up to 3 cores and 4GB of memory. This could easily be solved by directly installing Ubuntu 10.04 on a separate partition, or using a chroot to work in. The approach that was taken was using a newer virtualization technology called Linux Containers or LXC which provides isolation from the host machine while allowing the guest machine to utilize the entire memory and CPU cores. However, even with these improvements it was apparent that the test cases primarily used only a single CPU and did not take advantage of multiple cores.

Initial Challenges

Initially we wanted to run KLEE on an open source project. In exploring open source projects, we decided to investigate using a project called 'compiz' which is a C++ library used to render graphical windows on Linux distributions. It is a project known for having many defects reported, which is also one of the reasons why it was chosen. After running through the examples, it became clear that choosing a project that depended on many other libraries may be very problematic since it would require compiling and linking other libraries in LLVM and/or stubbing out complex library functions. Because of this we decided evaluate and reproduce the study using Coreutils but with a newer release.

Simple Examples

As part of our project we created some simple examples of programs with the types of errors detected by KLEE. These examples are small code snippets that do not do much besides show a possible fault detected in KLEE. All examples utilize the KLEE library and compile in the symbolic variables in the main method. We include the code as well as the error outputs from KLEE as an appendix to this paper.

Analysis of Examples

There were a few problems when generating the simple examples and running them. First, notice there are a significant number of warning messages in the output, even in a simple example. For a new user, this can be confusing and as it is unclear if the tool is actually running correctly and successfully.

In the first attempt to create the examples we used symbolic arguments from the command line. It quickly became evident that for this case, using the compiled in symbolic execution symbols would be both easier as well more exact since you could create different types of symbolic arguments, not just strings. One of the challenges found in moving between methods of specifying symbolic variables was that you also run the executables differently when running the test cases for applications using the command line symbolic arguments vs. symbolic variables compiled into the code.

Coreutils-8.16

Running KLEE with a newer revision of Coreutils ended up being more problematic than anticipated. Although the author has made a wise choice in choosing the LLVM compiler bytecode to develop KLEE, KLEE has not kept up with the more recent compiler flags and options as LLVM and gcc-llvm have continued to be developed. However, it seems that development is still active and perhaps will catch up in the future.

Additionally, some of the features in the original KLEE tool which increased code coverage by simulating system call faults has been removed from the distributed version of KLEE. According to the paper, utilizing this feature of KLEE in some cases greatly increased the test coverage, however the revision of KLEE downloaded from the site did not have this capability [2].

For our project, we created scripts to setup the download the tool, setup the environment, compile the project, and run simple examples as well as the coreutils example. Updating to the newer Coreutils-8.16 required modifying some of the files downloaded, and any patches required to make it work were also automated in the scripts. These scripts are available on github at the following URL:
`https://github.com/arges/vvklee` .

Part of this listing is contained in the appendix.

4. RESULTS

Coreutils Testing

Coreutils was tested using the following snippet of code:

```
for i in $(ls *.bc ); do
    run_test ${i%.bc} "--sym-args 0 8 8 --sym-files 2 8" 0
done
```

This essentially ran through every program instrumented with KLEE and used the POSIX test harness layer. The POSIX layer provides a method of tested with command line arguments and inputting files to test the particular program. The above arguments used indicate to test with at minimum 0 arguments to a maximum of 8 arguments with a size of 8 characters at maximum. In addition at most 2 files are created with 8 characters in the files to test the program. Although this resulted in some coverage of 100% in certain programs, in others the coverage wasn't as great. Figures 1-5 show the results of these tests, the code coverage from gcov, and the instruction and branch coverage as reported by KLEE. The execution time information collected revealed that most tests took the maximum specified time (1800s) to execute, which means that the symbolic execution for those particular parameters wasn't fully explored.

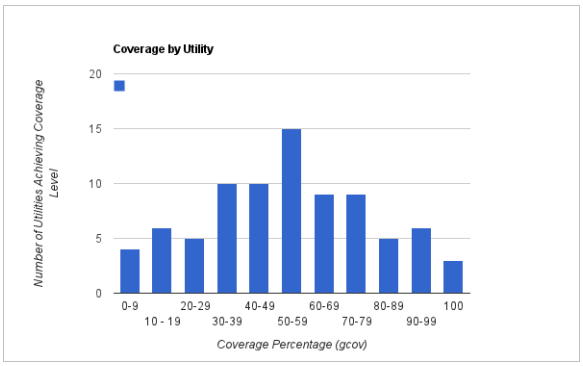

Figure 1: Coverage by Utility (Binned)

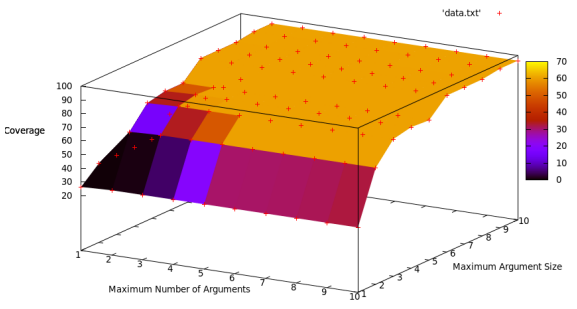

Figure 2: Maximum Arguments and Maximum Argument Size versus Code Coverage

Defects

KLEE was unable to find defects using the test cases that were run. This could be due to not running with large enough arguments to be able to detect an error condition in cases where coverage is not 100%. Perhaps more directed test are needed to achieve higher coverage and possibly detect defects.

Coverage Versus Argument Size

In addition to a full test on coreutils, the program 'echo' was chosen and run with various arguments to 'sym-args' to show coverage when varying these two variables. Figure 2 shows that varying the argument size and number of arguments both increase the coverage very rapidly. Because

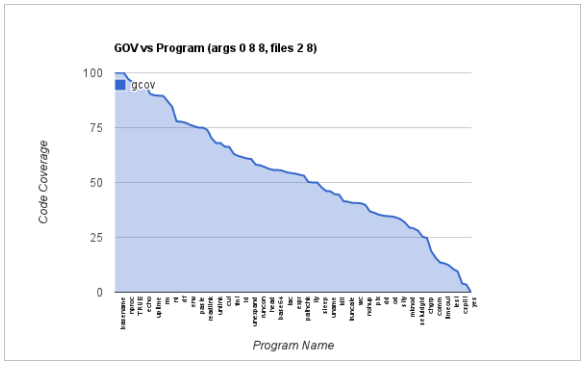

Figure 3: Program Name versus Code Coverage

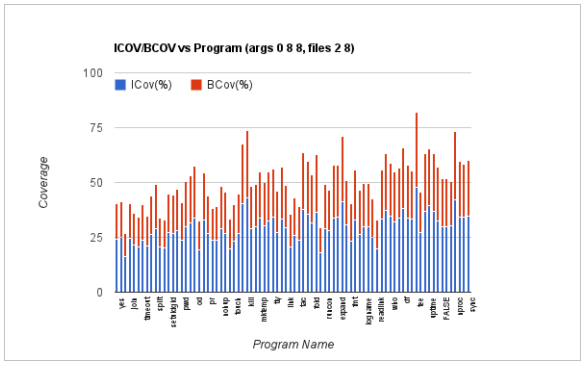

Figure 4: Program Name versus Instruction and Branch Coverage

this is dependent on the type of program sometimes the argument size could be more beneficial to increasing coverage vs number of arguments or vice versa. Echo has 5 different flags can it can be run with. It can be seen from this graph that after 5 arguments the coverage along this axis remains constant. However, increasing the argument size seems to start to level out around 6 arguments. From running this type of test it is evident that understanding the program structure is important to determining how KLEE should be used to run the analysis on the program to maximize the effectiveness.

5. ANALYSIS

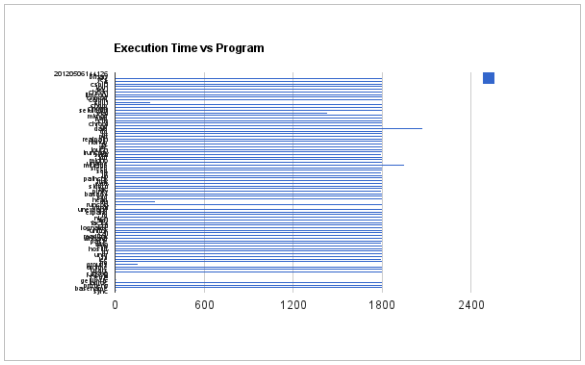

Figure 5: Program Name versus Execution Time

The tests resulted in 53.67% code coverage on average, with 3 utilities achieving 100% coverage and took 37 hours to complete execution. Because KLEE as configured using a time-out value of 1800 seconds, many of the tests took the maximum time to run and therefore the actual complete search did not complete. These tests then took the maximum time which makes sense that the complete tests took so long. Instruction coverage as reported by the KLEE tool was 31.21% on average. Branch coverage was 21.59% on average as reported by the KLEE tool.

Limitations

From our observations, the most useful improvement would be enabling parallel execution. The lack of support for distributing the constraint solving to multiple cores or computers has also been noted on the KLEE website as a future improvement as well as by users on the still active mailing list. Depending on the application, the high execution time could be a very limiting factor to adoption. In industry, with the push towards reduced development cycles, continuous integration, automated regular builds and regression test suites, it would be desirable to have this type of analysis done on a regular basis to catch any faults that may be introduced. Additionally, introducing parallelism may allow runs of much larger applications to be more feasible.

The next major category of limitations would be in the area of usability. Making the tool easier to setup, easily view the results and not provide false positive information. The tutorials on the website were good, however offered little help beyond the basic example. Many of the challenges noted in the ACM article, âĂIJA few billion lines of code later: using static analysis to find bugs in the real worldâĂİ would have to be addressed before this type of tool could achieve widespread use [1].

Although it appeared in some documentation on the KLEE website and paper that you could stub out library functions in your application, there was very little documentation on how to actually do this. Since the program is open source, the actual code would have to be analyzed to figure it out and implement other stub functions. As the tool stands now, the best application of KLEE would be something small, written in native C and that doesn't use many external complex libraries. If the tool improves in the future, and it becomes more user-friendly, it may be more easily added to fundamental libraries and in turn higher level applications could take advantage of this type of technology. Overall, this rigor in testing is sorely needed by many projects and the prospects of a easy to use tool are very exciting.

6. FUTURE WORK

KLEE seems to have potential however there are still a fair amount of steps required to get it running and analyze code. Although it was documented decently, there still are improvements that could be made to increase adoption of the tool. Future work could include continuing to use KLEE on another project and understanding how to use the tool features to stub out library functions.

Rather than having KLEE analyze the entire application, it might be more useful with the current limitations on computation to utilize it as a unit test tool to test smaller parts of larger applications. This would also help to possibly control the state explosion problem.

Use of KLEE requires a separate build of the application into LLVM bytecode. When integrating into an existing system, the build would have to be modified to accommodate this as well as changes may have to be made to the build flags used depending on if the gcc-llvm compiler supports them.

Output from these tools requires quite a bit of commands and searching. Tools that graph results and provide better highlighting of errors and test cases would also make this easier and friendlier for developers. Overall, the work is exciting but there is some work to make this a viable tool for industry as well as typical developers.

7. CONCLUSION

Overall, KLEE has potential to be very useful in detecting faults that are very difficult to reproduce. The tool itself, once you run it has some outputs that can be confusing at first, however the power in being able to generate a test case and replay it on the program is very powerful. KLEE's most significant limitation is with respect to scalability followed closely by usability. It appears to be possible to stub out libraries or parts of an application that you do not want to test, however the capability to do this is not well documented. In reproducing results from the KLEE run on Coreutils, we were unable to achieve as high of coverage as the original paper claimed. In running with the new version of Coreutils 8-16, we were unable to find any additional defects that were introduced.

8. REFERENCES

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[2] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[3] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1066 –1071, may 2011.

[4] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, 2003.

[5] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

APPENDIX

A. MEMORY ACCESSES

Code:

```
/*************************************************************
*  This is a simple example demonstrating an error with KLEE
*    for a buffer overflow problem or pointer issue.
*************************************************************/
#include <string.h>
void copyBuffer(char * bof)
{
   char buf[8];
   // oh no! we did not check the buffer sizes we are copying
   strcpy(buf, bof);
}
int main (int argc, char * argv[])
{
   char myStr[20];
   int x;
   klee_make_symbolic(&myStr, sizeof(myStr), "myStr");
   // make a constraint that the first 7 characters are non-null
   // so that we will produce 3 different test cases, one with ane
   // error, one on the boundary condition and one with an error
   for (x=0; x< 7; x++)
   {
      klee_assume(myStr[x] != '\0');
   }
   copyBuffer(myStr);
   return 0;
}
```

KLEE Output:

```
KLEE: NOTE: Using model: /home/jenny/klee/Release+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-0"
WARNING: While resolving call to function '__user_main' arguments were dropped!
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: klee_get_valuel
KLEE: WARNING: calling external: syscall(16, 0, 21505, 52704704)
KLEE: ERROR: strcpy.c/strcpy.c:27: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 5536
KLEE: done: completed paths = 42
KLEE: done: generated tests = 4
Output Error File: test0001.ptr.err
Error: memory error: out of bound pointer
File: strcpy.c/strcpy.c
Line: 27
Stack:
        #0 00001071 in __user_main (argc=5, argv=55567248) at strcpy.c/strcpy.c:27
        #1 00001462 in main (=5, =55567248)
Info:
        address: 53127588
        next: object at 53138272 of size 41
                MO56[41] allocated at global:.str7
        prev: object at 53127568 of size 20
                MO582[20] allocated at __user_main():  %myStr = alloca [20 x i8], align 1
Ktest generated test case info:
ktest file : 'test000001.ktest'
args       : ['./memoryAccess.o', '--sym-args', '0', '2', '4']
num objects: 3
object    0: name: 'n_args'
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'
object    2: name: 'myStr'
object    2: size: 20
object    2: data: '\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x02'
Coverage info: 100%
```

B. CALL TO ABORT

Code:

```
/*********************************************************************
*  This is a simple example demonstrating an error with KLEE
*    for a call to abort.
*********************************************************************/
#include <string.h>
#include <stdlib.h>
int main (int argc, char * argv[])
{
   char myStr[5];
   klee_make_symbolic(&myStr, sizeof(myStr), "myStr");
   if(strcmp(myStr, "bye!") == 0)
   {
       // someone said bye, time to go!
       abort();
   }
   return 0;
}
```

KLEE Output:

```
KLEE: NOTE: Using model: /home/jenny/klee/Release+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-0"
WARNING: While resolving call to function '__user_main' arguments were dropped!
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: klee_get_valuel
KLEE: WARNING: calling external: syscall(16, 0, 21505, 54621008)
KLEE: ERROR: abort.c/abort.c:12: abort failure
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 3967
KLEE: done: completed paths = 15
KLEE: done: generated tests = 2
Output Error File: test0002.abort.err
KLEE: NOTE: Using model: /home/jenny/klee/Release+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-0"
WARNING: While resolving call to function '__user_main' arguments were dropped!
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: klee_get_valuel
KLEE: WARNING: calling external: syscall(16, 0, 21505, 54621008)
KLEE: ERROR: abort.c/abort.c:12: abort failure
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 3967
KLEE: done: completed paths = 15
KLEE: done: generated tests = 2
Ktest generated test case info:
ktest file : 'test000002.ktest'
args       : ['./abort.o', '--sym-args', '0', '2', '4']
num objects: 3
object    0: name: 'n_args'
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'
object    2: name: 'myStr'
object    2: size: 5
object    2: data: 'bye!\x00'
Coverage info: 80%
```

C. MULTIPLE CALLS TO FREE

Code:

```
/*************************************************************
*  This is a simple example demonstrating an error with KLEE
*    for multiple calls to free.
*************************************************************/
int doSomething(int x, void * buf)
{
   if (x%3 == 2)
   {
      free((void*)buf);
      return 0;
   }
   return 1;
}
int main (int argc, char * argv[])
{
   int x;
   int * myBuf = (int *)malloc(sizeof(int)*20);
   klee_make_symbolic(&x, sizeof(x), "x");
   if (!doSomething(x, myBuf))
   {
      free((void *)myBuf);
   }
   return 0;
}
```

KLEE Output:

```
KLEE: NOTE: Using model: /home/jenny/klee/Release+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-0"
WARNING: While resolving call to function '__user_main' arguments were dropped!
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: klee_get_valuel
KLEE: WARNING: calling external: syscall(16, 0, 21505, 28435360)
KLEE: ERROR: free.c/free.c:27: memory error: invalid pointer: free
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 3496
KLEE: done: completed paths = 6
KLEE: done: generated tests = 2
Output Error File: test0002.ptr.err
Error: memory error: invalid pointer: free
File: free.c/free.c
Line: 27
Stack:
        #0 00001069 in __user_main (argc=5, argv=27883552) at free.c/free.c:27
        #1 00001456 in main (=5, =27883552)
Info:
        address: 26150864
        next: object at 26166208 of size 11
                MO9[11] allocated at main():  %k_termios.i.i1.i.i.i = alloca %struct.__kernel_termios, align 8
        prev: object at 26142512 of size 23
                MO26[23] allocated at main():  %k_termios.i.i1.i.i.i = alloca %struct.__kernel_termios, align 8
Ktest generated test case info:
ktest file : 'test000002.ktest'
args       : ['./free.o', '--sym-args', '0', '2', '4']
num objects: 3
object    0: name: 'n_args'
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'
object    2: name: 'x'
object    2: size: 4
```

```
object    2: data: '\xe8F\x80\x7f'
Coverage info: 72.73%
```

Assertion

Code:

```
/*************************************************************
*  This is a simple example demonstrating an error with KLEE
*    for a true assert statement.
*************************************************************/
#include <assert.h>
void checkValidRange(int a, int b)
{
   assert(a == 2*b);
}
int main (int argc, char * argv[])
{
  int myInt = 0;
  int yourInt = 0;
  klee_make_symbolic(&myInt, sizeof(myInt), "myInt");
  klee_make_symbolic(&yourInt, sizeof(yourInt), "yourInt");
  checkValidRange(myInt, yourInt);
  return 0;
}
```

KLEE Output:

```
KLEE: NOTE: Using model: /home/jenny/klee/Release+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-0"
WARNING: While resolving call to function '__user_main' arguments were dropped!
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: klee_get_valuel
KLEE: WARNING: calling external: syscall(16, 0, 21505, 41744624)
KLEE: ERROR: assert.c/assert.c:9: ASSERTION FAIL: a == 2*b
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 3493
KLEE: done: completed paths = 6
KLEE: done: generated tests = 3
Output Error File: test0001.assert.err
Error: ASSERTION FAIL: a == 2*b
File: assert.c/assert.c
Line: 9
Stack:
        #0 00001072 in __user_main (argc=5, argv=39224000) at assert.c/assert.c:9
        #1 00001459 in main (=5, =39224000)
Ktest generated test case info:
ktest file : 'test000001.ktest'
args       : ['./assert.o', '--sym-args', '0', '2', '4']
num objects: 4
object    0: name: 'n_args'
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'
object    2: name: 'myInt'
object    2: size: 4
object    2: data: '\x02\x00\x00\x00'
object    3: name: 'yourInt'
object    3: size: 4
object    3: data: '\xfe\xff\xff\x7f'
Coverage info: 0%, KLEE had problems replaying the assert test case.
```

D. DIVIDE BY ZERO

Code:

```
/*************************************************************
```

```
*  This is a simple example demonstrating an error with KLEE
*    for a divide by zero error.
*************************************************************/
#include <string.h>
int main (int argc, char * argv[])
{
  int myInt = 0;
  int yourInt = 0;
  klee_make_symbolic(&yourInt, sizeof(yourInt), "yourInt");
  klee_make_symbolic(&myInt, sizeof(myInt), "myInt");
  myInt = yourInt-1;
  // this will cause a divide by zero sometimes
  return yourInt/myInt;
}
KLEE Output:
KLEE: NOTE: Using model: /home/jenny/klee/Release+Asserts/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-0"
WARNING: While resolving call to function '__user_main' arguments were dropped!
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: klee_get_valuel
KLEE: WARNING: calling external: syscall(16, 0, 21505, 38920912)
KLEE: ERROR: klee_div_zero_check.c/klee_div_zero_check.c:14: divide by zero
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 3505
KLEE: done: completed paths = 6
KLEE: done: generated tests = 2
Output Error File: test0002.div.err
Error: divide by zero
File: klee_div_zero_check.c/klee_div_zero_check.c
Line: 14
Stack:
        #0 00002296 in klee_div_zero_check (z) at klee_div_zero_check.c/klee_div_zero_check.c:14
        #1 00001067 in __user_main (argc=5, argv=39289200) at divideZero.c/divideZero.c:16
        #2 00001452 in main (=5, =39289200)
Ktest generated test case info:
ktest file : 'test000002.ktest'
args       : ['./divideZero.o', '--sym-args', '0', '2', '4']
num objects: 4
object    0: name: 'n_args'
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'
object    2: name: 'yourInt'
object    2: size: 4
object    2: data: '\x01\x00\x00\x00'
object    3: name: 'myInt'
object    3: size: 4
object    3: data: '\x00\x00\x00\x00'
Coverage info: 100%
```

E. BUILD SCRIPT FOR STUDY

```
#!/bin/bash
#
# Script to build and setup environment for our V&V project.
#
# (C)2012 by Chris J Arges <christopherarges@gmail.com>
#

# Right now this has only been tested on 32-bit Lucid.
# Versions of llvm-gcc > gcc4.2/llvm-2.8 don't have the --emit-llvm option.
```

```
MAKEOPTS="-j8"
CWD=$(pwd)

KLEE_DIR="${CWD}/build/klee"
KLEE_PATH="${CWD}/build/klee/scripts/klee-gcc"
KLEE_UCLIBC_FILE="klee-uclibc-0.02-i386.tgz"

CDE_BUILD=
KLEE_CDE_PACKAGE="http://keeda.stanford.edu/~pgbovine/klee-cde-package.v2.tar.bz2"
KLEE_POSTFIX=""

function download() {
        echo "Downloading and installing required files."
        sudo apt-get install --force-yes build-essential wget autoconf automake

        # If custom build download/extact all files.
        if [ ! -n "$CDE_BUILD" ]; then
                mkdir build
                cd build

                # get dependencies
                sudo apt-get install --force-yes subversion \
                        kcachegrind llvm-gcc-4.2
                sudo apt-get build-dep --force-yes llvm

                # llvm-2.8
                wget http://llvm.org/releases/2.8/llvm-2.8.tgz
                tar -xf llvm-2.8.tgz
                # uclibc-0.02 klee
                wget http://www.doc.ic.ac.uk/~cristic/klee/${KLEE_UCLIBC_FILE}
                tar -xf ${KLEE_UCLIBC_FILE}
                # klee from svn
                svn co http://llvm.org/svn/llvm-project/klee/trunk klee

                cd ..
        else
                # download CDE package (if selected)
                if [ ! -e klee-cde-package.v2.tar.bz2 ]; then
                         wget ${KLEE_CDE_PACKAGE}
                fi
                if [ ! -d klee-cde-package ]; then
                        tar -xf klee-cde-package.v2.tar.bz2
                fi
        fi
}

function build() {
        if [ ! -n "$CDE_BUILD" ]; then
                echo "Building projects"

                cd build

                # build llvm
                cd llvm-2.8
                LLVM_PATH=$(pwd)
                ./configure --enable-optimized --enable-assertions
                make ${MAKEOPTS} || fail "make llvm failed!"
                cd ..

                # build klee-uclibc
                cd klee-uclibc-0.02-i386
                ./configure --with-llvm=${LLVM_PATH}
```

```
                make ${MAKEOPTS} || fail "make klee-uclibc failed!"
                UCLIBC_PATH=$(pwd)
                cd ..

                # build klee
                cd klee
                ./configure --with-llvm=${LLVM_PATH} --with-uclibc=${UCLIBC_PATH} --enable-posix-runtime
                make ENABLED_OPTIMIZED=1 ${MAKEOPTS} || fail "make klee failed!"
                make check
                make unittests
                cd ../..
        else
                echo "Don't need to build anything for CDE"
        fi
        win
}

COREUTILS_VER=8.16
COREUTILS_ARCHIVE=coreutils-${COREUTILS_VER}.tar.xz
COREUTILS_DIR=coreutils-${COREUTILS_VER}
COREUTILS_URL=http://ftp.gnu.org/gnu/coreutils/${COREUTILS_ARCHIVE}

function example() {
        echo "Building coreutils example"
        mkdir -p coreutils-klee && cd coreutils-klee

        # download and extract if it doesn't exist
        if [ ! -e ${COREUTILS_ARCHIVE} ]; then wget ${COREUTILS_URL}; fi
        if [ ! -d ${COREUTILS_DIR} ]; then tar -xf ${COREUTILS_ARCHIVE}; fi
        cd ${COREUTILS_DIR}

        # patch system
        for i in $(ls ../../patch); do
                patch -p1 < ../../patch/$i
        done

        # build gcov tests
        mkdir -p obj-gcov && cd obj-gcov
        ../configure --disable-nls CFLAGS="-g -fprofile-arcs -ftest-coverage"
        make ${MAKEOPTS} || fail "Couldn't build obj-gcov coreutils!"
        cd ..

        # build llvm objects
        mkdir -p obj-llvm && cd obj-llvm
        ../configure --disable-nls CFLAGS="-g"
        make CC=${KLEE_PATH} CPPFLAGS="-std=gnu99" ${MAKEOPTS} || fail "Couldn't build obj-llvm coreutils!"

        # test it
        #${KLEE_DIR}/Release+Asserts/bin/klee --libc=uclibc --posix-runtime src/cat.bc --version || fail "Couldn't run

        cd ../..
        win
}

function fail() {
echo "     FAIL WHALE! "
echo ""
echo "W     W      W        "
echo "W        W  W     W   "
echo "              '.  W    "
echo "  .-\"\"-._     \ \.--|  "
echo " /       \"-..__) .-'   "
echo "|     _         /      "
```

```
echo "\'-.__,   .__.,\'       "
echo " \`\'----\'._\--\'      "
echo "VVVVVVVVVVVVVVVVVVVVV"
echo $1
exit 1
}

function win() {
echo "Huzzah! It works! :)"
}

while getopts "cdbhe?" opt; do
        case $opt in
        c)
                CDE_BUILD="yes"
                KLEE_POSTFIX=".cde"
                KLEE_PATH="$(pwd)/klee-cde-package/bin"
                KLEE_PATH="${CWD}/klee-cde-package/cde-root/home/pgbovine/klee/scripts/klee-gcc"
                ;;
        d)      download ;;
        b)      build ;;
        e)      example ;;
        *)      echo "Usage: $0 -dbe"
                echo "  -d download files"
                echo "  -b build files"
                echo "  -e build coreutils example"
                ;;
        esac
done
```

F. TEST SCRIPT FOR STUDY

```
#!/bin/bash
#
# Script to run coreutils tests and output proper data.
#
# Works with both CDE package or custom built package.
#
# (C)2012 by Chris J Arges <christopherarges@gmail.com>
#

CWD=$(pwd)
TARGET_DIR=${CWD}/coreutils-klee/coreutils-8.16/
LLVM_DIR=${TARGET_DIR}/obj-llvm
GCOV_DIR=${TARGET_DIR}/obj-gcov

TIMESTAMP=$(date +%Y%m%d%H%M%S)
RESULT_DIR=${CWD}/results/$TIMESTAMP

KLEE_BIN="klee${KLEE_POSTFIX}"
KLEE_STATS_BIN="klee-stats${KLEE_POSTFIX}"
KTEST_TOOL_BIN="ktest-tool${KLEE_POSTFIX}"
KLEE_REPLAY_BIN="klee-replay${KLEE_POSTFIX}"
KLEE_PATH="$(pwd)/build/klee/Release+Asserts/bin/"

#KLEE_ARGS="--only-output-states-covering-new --optimize --use-forked-stp \
#               --libc=uclibc --max-time=1800 --posix-runtime"

KLEE_ARGS="--simplify-sym-indices --max-memory=12288 --use-cex-cache \
        --disable-inlining --allow-external-sym-calls --watchdog \
        --max-memory-inhibit=false --only-output-states-covering-new \
        --optimize --use-forked-stp --libc=uclibc \
        --max-time=60 --posix-runtime"
```

```
function cleanup() {
        # Remove any gcov generated files.
        cd $GCOV_DIR/src
        rm -f *.gcda

        # Remove any klee directories.
        cd $LLVM_DIR/src
        rm -rf klee-*
}

# run_test <binary_name> <posix arguments> <klee arguments> <test #>
function run_test() {
        BINARY=$1
        ARGS=$2
        TESTNO=$3

        # setup the test
        echo ">>> testing $BINARY : $ARGS"
        cleanup
        START=$(date +%s)
        TEST_DIR=${RESULT_DIR}/${BINARY}/${TESTNO}/
        mkdir -p ${TEST_DIR}

        # run klee coverage
        ${KLEE_BIN} ${KLEE_ARGS} ./${BINARY}.bc ${ARGS} &> ${TEST_DIR}/klee.log
        ${KLEE_STATS_BIN} klee-last &> ${TEST_DIR}/klee-stats.log
        ${KTEST_TOOL_BIN} klee-last/*.ktest &> ${TEST_DIR}/ktest-tool.log
        cp -Lr klee-last ${TEST_DIR}/

        # calculate coverage of test
        cd $GCOV_DIR/src
        ${KLEE_REPLAY_BIN} ./${BINARY} ${TEST_DIR}/klee-last/*.ktest &> ${TEST_DIR}/klee-replay.log
        gcov ./${BINARY} &> ${TEST_DIR}/gcov.log

        # keep track of time
        END=$(date +%s)
        DIFF=$(( $END - $START ))
        echo "  execution time: $DIFF s" | tee ${TEST_DIR}/time.log

        # show interesting stuff on the screen
        egrep "(executed|File)" ${TEST_DIR}/gcov.log | cut -f 2 -d ':'
        echo ""
}

# add KLEE bin to the path
add_to_path() {
        if [ -d "${KLEE_PATH}" ] && [[ ":$PATH:" != *":${KLEE_PATH}:"* ]]; then
                PATH="$PATH:${KLEE_PATH}"
        fi
}

function run_tests() {
        echo $1 $2 $3 $4 $5

        echo ${RESULT_DIR}
        mkdir -p ${RESULT_DIR}
        add_to_path
        echo $1 > ${RESULT_DIR}/type
        cd $LLVM_DIR/src

        case $1 in
        "small")
                for i in $(ls *.bc ); do
```

```
                        run_test ${i%.bc} "--sym-arg 3" 0
                done
                ;;
        "medium")
                for i in $(ls *.bc ); do
                        run_test ${i%.bc} "--sym-args 0 2 4 --sym-files 2 2" 0
                done
                ;;
        "large")
                for i in $(ls *.bc ); do
                        run_test ${i%.bc} "--sym-args 0 8 8 --sym-files 2 8" 0
                done
                ;;
        "args")
                for x in {1..10}; do
                        for y in {1..10}; do
                                run_test echo "--sym-args 0 ${x} ${y}" ${x}_${y}
                        done
                done
                ;;
        "other")
                run_test $2 "$3 $4"
                ;;
        esac
}

while getopts "smlaco:" opt; do
        case $opt in
        s)      run_tests "small";;
        m)      run_tests "medium";;
        l)      run_tests "large";;
        a)      run_tests "args";;
        o)      run_tests "other" $OPTARG;;
        c)
                KLEE_POSTFIX=".cde"
                KLEE_PATH="$(pwd)/klee-cde-package/bin"
                KLEE_BIN="klee${KLEE_POSTFIX}"
                KLEE_STATS_BIN="klee-stats${KLEE_POSTFIX}"
                KTEST_TOOL_BIN="ktest-tool${KLEE_POSTFIX}"
                KLEE_REPLAY_BIN="klee-replay${KLEE_POSTFIX}"
                ;;
        \?)     echo "Usage: $0"
                echo "  -s      run SMALL test."
                echo "  -m      run MEDIUM test."
                echo "  -l      run LARGE test (run overnight!)."
                echo "  -a      run args comparison test."
                echo "  -c      use CDE package."
                ;;
        esac
done
```