Dynamic Symbolic Execution using KLEE

Geethu Girish[20985805], Mareena Francis[20985107], and Shanthi Meena Arumugam[20986268]

[1] Master of Engineering, University of Waterloo
[2] ggirish@uwaterloo.ca
[3] m22francis@uwaterloo.ca
[4] smarumug@uwaterloo.ca

Abstract. The main challenge in ensuring a program is correct is the selection of effective verification technique that can state the program to be free of errors. Manual and adhoc testing techniques can verify the correctness of a program to some extent. However, these testing methods are not sufficient to confirm that a program will work for all the possible inputs and is not prone to errors. Dynamic Symbolic Execution is a highly applicable program analysis technique that is used to cover execution paths in a program by executing the program symbolically and reasoning the constraints obtained through the exploration of feasible execution paths. In this project a Dynamic Symbolic Execution tool called KLEE is used for executing three open source projects and tries to achieve maximum coverage in order to detect program errors.

Keywords: Dynamic Symbolic Execution · KLEE · Coverage.

1 Introduction

This project report aims to record and analyze the results of Dynamic Symbolic Execution performed on three different open source projects using KLEE. KLEE is a Symbolic Execution tool used to reach every executable line of the selected program and detects different types of failures that causes abnormal program behavior. In addition to this, KLEE generates test cases that cover both the discovered execution paths and paths that result in failure.

The upcoming sections illustrate basic principles of Symbolic Execution and how it is used for program analysis and failure detection. Section 3 includes an overview of KLEE architecture and implementation of Dynamic Symbolic Execution on three open source projects followed by an analysis section which records the output generated by KLEE execution and analysis of results obtained.

2 Theoretical Foundation

The key principle behind symbolic execution is to perform the software testing by using symbolic variables for program evaluation and to represent the values of program variables as symbolic expressions [1]. Dynamic Symbolic Execution(DSE) is an extension of Symbolic Execution which executes the program using specific inputs. Here, concrete execution guides the symbolic execution and stores a concrete state in addition to the symbolic state and path conditions. During the process of collecting constraints, the input variables are sequentially symbolized, and then the constraint solver is used to judge the change of input, and then the next execution path of the program is decided [2]. If the program being tested includes a branch operation, the constraints are reversed to generate a new path condition which explores the other branch of the program. This process is repeated for each branch in order to execute all the possible paths symbolically and to generate test cases for all the explored paths.

2.1 Constraint Solving in Dynamic Symbolic Execution using KLEE

Constraint solvers are decision procedures for problems expressed in logical formulas: for instance, the boolean satisfiability problem (also known as SAT) aims at determining whether there exists an interpretation of the symbols of a formula that makes it true [3].

The symbolic execution proceeds by exploring the program execution paths and recording the path conditions by appending it to the current constraints to generate new constraints to be solved by the constraint solver.

2.2 Generating inputs for coverage

Dynamic Symbolic Execution begins with choosing an arbitrary input, this can be feeded from a fuzzer which generates random inputs or the inputs can be provided by the user. Once the arbitrary input is chosen, the program will be executed concretely and symbolically in parallel. The concrete execution takes the program execution to proceed to a particular branch and symbolic execution follows the same path. The branch conditions which lead to that branch will be added to the path condition. To explore other paths, the path conditions are flipped and execute the branch symbolically by invoking the constraint solver to generate new inputs that satisfy the new constraints. The process of executing the program concretely and symbolically in parallel is repeated as many times in order to explore all possible execution paths.

2.3 Remarkable advantages of Dynamic Symbolic Execution

Dynamic Symbolic Execution allows the implementation of concrete execution for the computations which are hard to execute symbolically. This makes the process of test case generation effective by allowing the complex computations to be a part of the program execution and allows the program to implement concrete and symbolic execution in parallel.

3 Methodologies

In our project, KLEE is used for dynamic symbolic execution to test our program.

3.1 Overview

A symbolic execution tool called KLEE can greatly outperform the developer's own hand-written test suites in terms of coverage. KLEE can provide automatically generated high-coverage test inputs that outperform manual and random testing methods, which have low performance. KLEE[6] wants to achieve two things. First, to run a program in every feasible method until every line of executable code has been reached. Additionally, each line of code that is reachable in a program should be tested against an input that fulfills the path constraint errors. In addition, KLEE produces test cases as opposed to manually creating test suites for issues that are discovered and have good code coverage.

3.2 Installation

In our project, we have used KLEE with Docker[5] installation as mentioned on the official KLEE website[5]. KLEE runs LLVM code and not C code.

Steps used for running KLEE inside a docker container[5] is listed below.

Step:1 Install Docker for Linux/Mac/Windows

Step:2 Clone the repo(Build a docker image locally) using below commands, or pull the docker from container

git clone https://github.com/klee/klee.git

cd klee docker build -t klee/klee .

Step:3 Creating a KLEE docker container:

docker run –rm -ti –ulimit='stack=-1:-1' klee/klee

The –ulimit option sets an unlimited stack size inside the container. This is to avoid stack overflow issues when running KLEE.

3.3 Usage

A user can start checking many real programs with KLEE in seconds: KLEE typically requires no source modifications or manual work. Users first compile their code to bytecode using the publicly-available clang compiler for C. We compiled our program using the following code[5]

3.4 Implementation

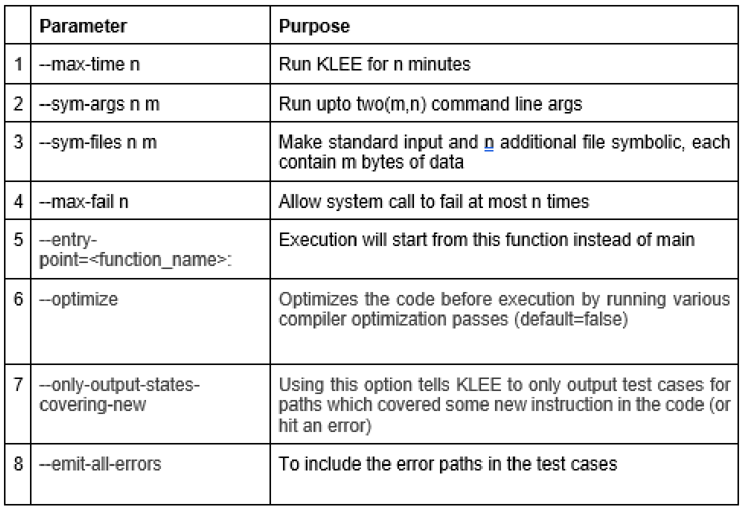
We are utilizing the KLEE Symbolic Execution Engine and their Docker image here on Mac/Windows. To inform the tool to utilize that as the "search space," we identify the inputs as symbolic. This indicates that KLEE verifies all potential inputs for the form that we supply it. This section explains how KLEE works by walking the reader through our open source git projects. We have selected 3 different programs in our projects and symbolically executed and verified with KLEE.

3.4.1 Open source project 1 - Tiny-Regex[7] The first project we used KLEE is tiny-regex which is a small and portable regular expression library in C. Compile and emit LLVM bitcode using KLEE following command

clang -emit-llvm -c -g -O0 tiny-regex.c

Then run KLEE on the generated bytecode, optionally stating the number, size, and type of symbolic inputs to test the code on as mentioned below, klee –libc=uclibc -max-memory=1000 -max-time=30 -only-output-states-covering-new re.bc

The first option, –max-time, tells KLEE to check tiny-regex.bc for at most two minutes. We can use different flags/parameters when running the klee command(see Fig.1)



	Parameter	Purpose
1	--max-time n	Run KLEE for n minutes
2	--sym-args n m	Run upto two(m,n) command line args
3	--sym-files n m	Make standard input and n additional file symbolic, each contain m bytes of data
4	--max-fail n	Allow system call to fail at most n times
5	--entry-point=<function_name>:	Execution will start from this function instead of main
6	--optimize	Optimizes the code before execution by running various compiler optimization passes (default=false)
7	--only-output-states-covering-new	Using this option tells KLEE to only output test cases for paths which covered some new instruction in the code (or hit an error)
8	--emit-all-errors	To include the error paths in the test cases

Fig. 1. Useful flags/parameters for KLEE execution

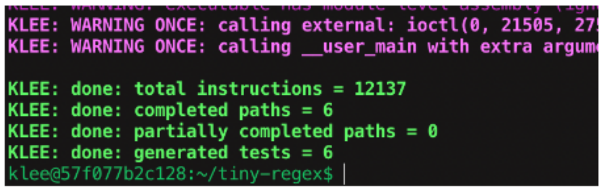
When running the test, variables are often symbolized by calling the Klee to make a symbolic function in the application.

We must use symbolic input while running this function through KLEE. The klee-make-symbolic() function (described in Klee/Klee.h) can be used to mark a variable as symbolic. It accepts three arguments: the address of the variable (memory location) we want to treat as symbolic, its size, and a name (which can be anything). Here is a straightforward main() method that calls get sign while designating the variable an as symbolic()

When klee calls the re-match method which is having 3 arguments, KLEE hits the branch argc greater than 1 at line 73, it uses its constraint solver STP [2] to see which directions can execute given the current path condition. For this branch, both directions are possible. KLEE forks execution and follows both paths, adding the constraint argc greater than 1 on the false path and argc less than or equal to 1 on the true path.

KLEE must choose which active path to follow first when there are several of them. Consider it follows the path that leads to the bug for the time being. KLEE forks five times (lines labeled with a "*"), and then expand accordingly. As it does so, it adds additional constraints to the contents of arg.

KLEE examines if any conceivable value made possible by the current path condition would result in an error at each risky action (such as a pointer dereference). KLEE finds no errors on the annotated route at line 81. The code then advances arg twice without confirming that the string has ended after taking the true branch, indicating that it has discovered input values that permit the read of arg to go outside of limits. If so, this increment points to faulty memory and skips the terminating "0"



```
KLEE: WARNING: executable has module level assembly (ign
KLEE: WARNING ONCE: calling external: ioctl(0, 21505, 27
KLEE: WARNING ONCE: calling __user_main with extra argum

KLEE: done: total instructions = 12137
KLEE: done: completed paths = 6
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 6
klee@57f077b2c128:~/tiny-regex$ 
```

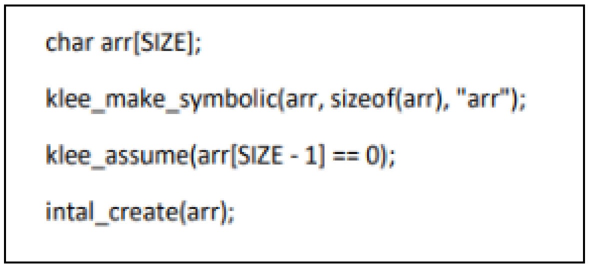
Fig. 2. Symbolic Execution output of method ismetachar()

KLEE output shows how many paths are explored for one small method named ismetachar(see Fig.2) Using the KLEE tool, this project has been verified and uncovered the hidden memory errors. *Execution files are attached in the git repo.

3.4.2 Open source project 2 - Integers of Arbitrary Length[4] This open source git project is a C library used to perform calculations on integers of arbitrary length. This program aims to generate integers of arbitrary length by invoking the method intal-create() by passing a character string of a non-negative integer provided in decimal digits. The program also includes other methods such as intal-add() which creates a new integer of arbitrary length and intal-destroy() which frees the memory allocated during creation of integer. The source code of the project is pulled from the open source Git repository [4] to the docker container.

The method intal-create() accepts a character string as input parameter. The Dynamic Symbolic Execution of this method proceeds by creating a symbolic character string in the main method of the program by calling the klee-make-symbolic function (see Fig.3). The code is then compiled using a clang compiler to generate bitcode and then execute it symbolically by using KLEE.

In the same way, another method of the program named append-char() executed symbolically. This method accepts one character string and one single



```
char arr[SIZE];

klee_make_symbolic(arr, sizeof(arr), "arr");

klee_assume(arr[SIZE - 1] == 0);

intal_create(arr);
```

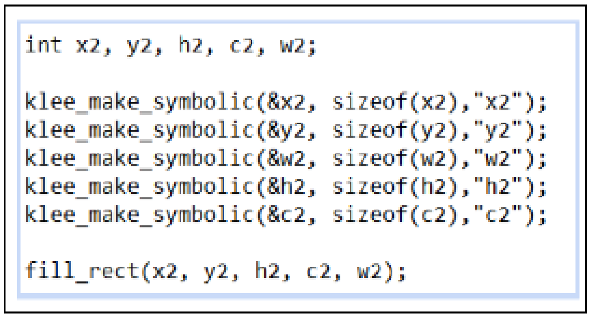
Fig. 3. Symbolic Execution of method intal-create()

character. Symbolic variables are created for these two parameters using klee-make-symbolic function by passing the address, size of variable and symbolic name. The method append-char() is then called by passing the symbolic variable instead of actual variables. Execution of the program using KLEE generates test cases for covering the paths. The generated test cases are stored in the klee output directory and also detects an error which is also stored in the output directory with extensions as .err. Test cases generated by KLEE can be opened in the user readable format by using the ktest-tool by specifying the test case file name.

3.4.3 Open source project 3 - Characters to Linux Buffer[8] The third project chosen is also an open source project in C which uses arrays of 0s and 1s to represent characters and numbers and these are displayed on screen using Linux Framebuffer routines. The method put-pixel() is used to assign color value except where memcpy() is used for which the value of color is already assigned into the element of the array. Inorder to proceed with the dynamic symbolic execution, each method is invoked individually. First method executed symbolically is fill-rect(). It takes 5 inputs which are integers x2, y2, h2, c2 and w2 and all the five variables are made symbolic using klee-make-symbolic() which takes 3 arguments, the address of the variable, size of the variable and a name for the variable. The code is shown in the below:

The code is then compiled using clang and executed symbolically. This generates a test result which illustrates the total number of instructions executed, number of generated test cases, completed paths and partial paths. The test cases information is stored in the files with extension .ktest inside the KLEE output directory generated after each execution.

Similarly another method with more parameters and different data types was executed symbolically. The method decdisp() accepts 10 parameters: one unsigned character array, one unsigned short variable, two unsigned integer variables and addresses of 6 unsigned short variables. All the variables were made symbolic using klee-make-symbolic() and used klee-assume() to give constraints for the array. (See fig.5)

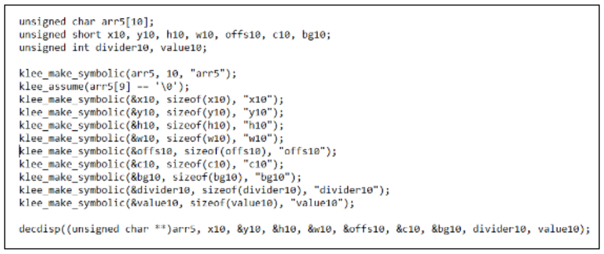


```
int x2, y2, h2, c2, w2;

klee_make_symbolic(&x2, sizeof(x2),"x2");
klee_make_symbolic(&y2, sizeof(y2),"y2");
klee_make_symbolic(&w2, sizeof(w2),"w2");
klee_make_symbolic(&h2, sizeof(h2),"h2");
klee_make_symbolic(&c2, sizeof(c2),"c2");

fill_rect(x2, y2, h2, c2, w2);
```

Fig. 4. Symbolic execution of method fill-rect()



```
unsigned char arr5[10];
unsigned short x10, y10, h10, w10, offs10, c10, bg10;
unsigned int divider10, value10;

klee_make_symbolic(arr5, 10, "arr5");
klee_assume(arr5[9] -- '\0');
klee_make_symbolic(&x10, sizeof(x10), "x10");
klee_make_symbolic(&y10, sizeof(y10), "y10");
klee_make_symbolic(&h10, sizeof(h10), "h10");
klee_make_symbolic(&w10, sizeof(w10), "w10");
klee_make_symbolic(&offs10, sizeof(offs10), "offs10");
klee_make_symbolic(&c10, sizeof(c10), "c10");
klee_make_symbolic(&bg10, sizeof(bg10), "bg10");
klee_make_symbolic(&divider10, sizeof(divider10), "divider10");
klee_make_symbolic(&value10, sizeof(value10), "value10");

decdisp((unsigned char **)arr5, x10, &y10, &h10, &w10, &offs10, &c10, &bg10, divider10, value10);
```

Fig. 5. Symbolic Execution output of method decdisp()

Execution of the program using KLEE generates test cases for covering the paths. The generated test cases are stored in the klee output directory and also detects three errors which are stored in the output directory with extensions as .err.

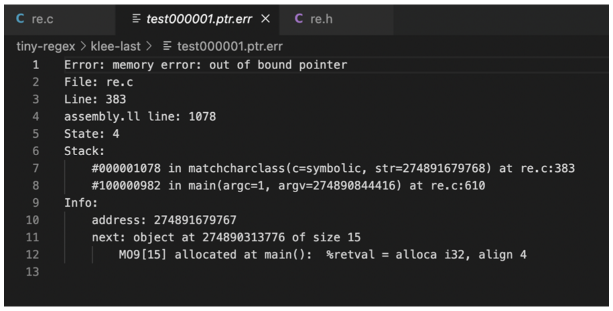
This code illustrates two additional common features. First, it has bugs, which KLEE finds and generates test cases for. Second, KLEE quickly achieves good code coverage: in two minutes it generates tests that cover all executable statements.

4 Analysis

4.1 4.1 Result Analysis

The following section records and analyses the obtained from KLEE execution.

4.1.1 Result Analysis of project 1 - Tiny-Regex When running the KLEE for the small method "matchcharclass", KLEE discovered the "memory error: out of bound pointer" when exploring the paths. The error image from the terminal is attached below for the reference(see Fig.6). When it enters the line no:383, KLEE determines that input values exist that allow the read of arg to go out of bounds: after taking the true value to the branch. After finding the error, KLEE will proceed to find the path by ignoring all the faulty path as we have provided the flag -only-output-states-covering-new re.bc while running with the time limit 30.

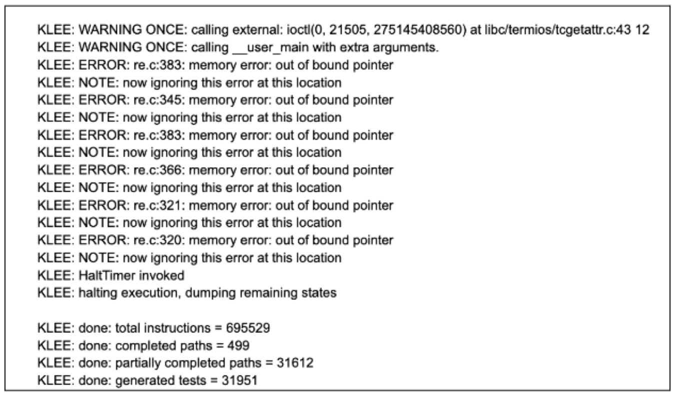


```
Error: memory error: out of bound pointer
File: re.c
Line: 383
assembly.ll line: 1078
State: 4
Stack:
    #000001078 in matchcharclass(c=symbolic, str=274891679768) at re.c:383
    #100000982 in main(argc=1, argv=274890844416) at re.c:610
Info:
    address: 274891679767
    next: object at 274890313776 of size 15
        MO9[15] allocated at main():  %retval = alloca i32, align 4
```

Fig. 6. Error detected in method matchcharclass()

As Klee continued to explore the paths, it explored almost 7000 paths for the entire program and generated almost lakhs of test cases in total for testing the different paths.

In total, it found 6 memory errors - out of bound errors in this program and 31000 paths. Because of the error, KLEE completed 499 paths and dumped all other paths which had errors. It results in a partially completed path with 31612.



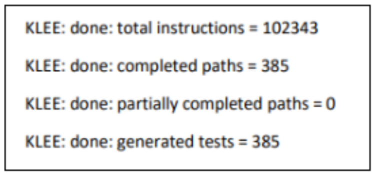
```
KLEE: WARNING ONCE: calling external: ioctl(0, 21505, 275145408560) at libc/termios/tcgetattr.c:43 12
KLEE: WARNING ONCE: calling __user_main with extra arguments.
KLEE: ERROR: re.c:383: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: re.c:345: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: re.c:383: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: re.c:366: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: re.c:321: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: re.c:320: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: HaltTimer invoked
KLEE: halting execution, dumping remaining states

KLEE: done: total instructions = 695529
KLEE: done: completed paths = 499
KLEE: done: partially completed paths = 31612
KLEE: done: generated tests = 31951
```

Fig. 7. List of errors and output generated by KLEE for Tiny-Regex

Klee code and output files are available in the git repository for reference.

4.1.2 Result Analysis of project 2 - Intal When KLEE is executed with the Integers of Arbitrary Length project, it generates test cases and traverses the program to cover paths based on the method executed. Symbolic execution of the method intal-create() generates a total of 385 test cases and all the execution paths corresponding to these test cases are covered which is indicated by the number of completed paths in the KLEE output. The output illustrates the total number of instructions executed, number of generated test cases, completed paths and partial paths (see Fig.8). The test cases information (see Fig.9) is stored in the files with extension .ktest inside the KLEE output directory generated after each execution. The number of partially completed paths is 0 which indicates there are no errors detected by KLEE for this method and it covers all the paths corresponding to the test cases generated.



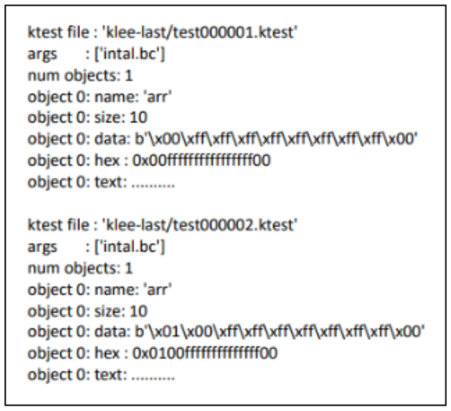
```
KLEE: done: total instructions = 102343

KLEE: done: completed paths = 385

KLEE: done: partially completed paths = 0

KLEE: done: generated tests = 385
```

Fig. 8. KLEE output for method intal-create()

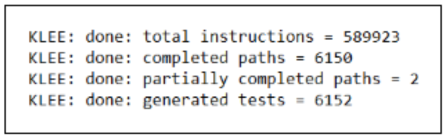


```
ktest file : 'klee-last/test000001.ktest'
args       : ['intal.bc']
num objects: 1
object 0: name: 'arr'
object 0: size: 10
object 0: data: b'\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00'
object 0: hex : 0x00ffffffffffffffff00
object 0: text: ..........

ktest file : 'klee-last/test000002.ktest'
args       : ['intal.bc']
num objects: 1
object 0: name: 'arr'
object 0: size: 10
object 0: data: b'\x01\x00\xff\xff\xff\xff\xff\xff\xff\x00'
object 0: hex : 0x0100ffffffffffffff00
object 0: text: ..........
```

Fig. 9. Few test cases generated by KLEE for method intal-create()

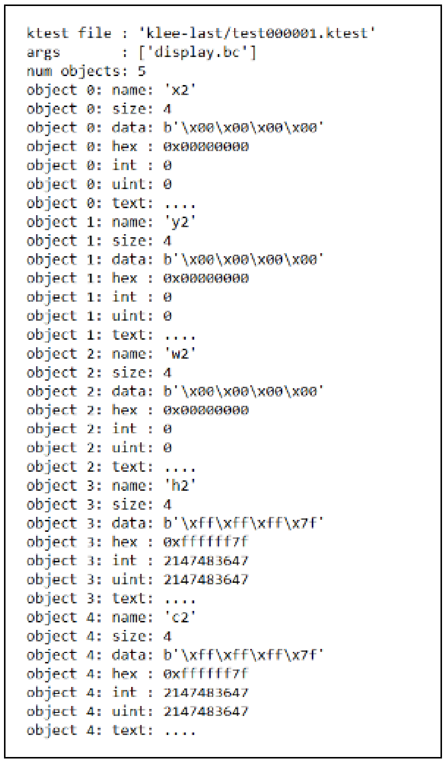
4.1.3 Result Analysis of project 3 - Characters to Linux Buffer the hard way When the fill-rect() method of display.c is executed symbolically, it generated 6152 test cases out of which 6150 paths are completely executed and



```
KLEE: done: total instructions = 589923
KLEE: done: completed paths = 6150
KLEE: done: partially completed paths = 2
KLEE: done: generated tests = 6152
```

Fig. 10. KLEE output for the method fill-rect()

2 paths are partially completed. The two paths which are partially completed indicate that there are errors which were found on the execution of these test cases. The output and the test cases which threw error are shown in Fig.10 and Fig.11 respectively. The output is stored in klee-out file and the result of the latest execution can be found in klee-last. While executing the fill-rect() function symbolically 5 integer variables were made symbolic.

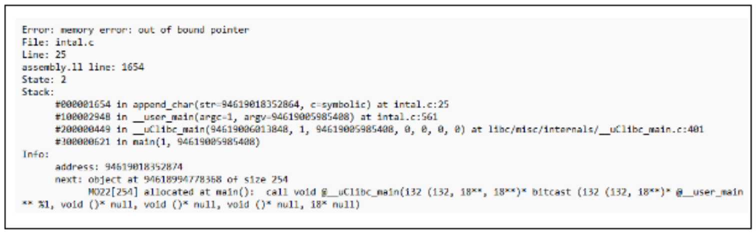


```
ktest file : 'klee-last/test000001.ktest'
args       : ['display.bc']
num objects: 5
object 0: name: 'x2'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
object 1: name: 'y2'
object 1: size: 4
object 1: data: b'\x00\x00\x00\x00'
object 1: hex : 0x00000000
object 1: int : 0
object 1: uint: 0
object 1: text: ....
object 2: name: 'w2'
object 2: size: 4
object 2: data: b'\x00\x00\x00\x00'
object 2: hex : 0x00000000
object 2: int : 0
object 2: uint: 0
object 2: text: ....
object 3: name: 'h2'
object 3: size: 4
object 3: data: b'\xff\xff\xff\x7f'
object 3: hex : 0xffffff7f
object 3: int : 2147483647
object 3: uint: 2147483647
object 3: text: ....
object 4: name: 'c2'
object 4: size: 4
object 4: data: b'\xff\xff\xff\x7f'
object 4: hex : 0xffffff7f
object 4: int : 2147483647
object 4: uint: 2147483647
object 4: text: ....
```

Fig. 11. Test case that hits error at line 81

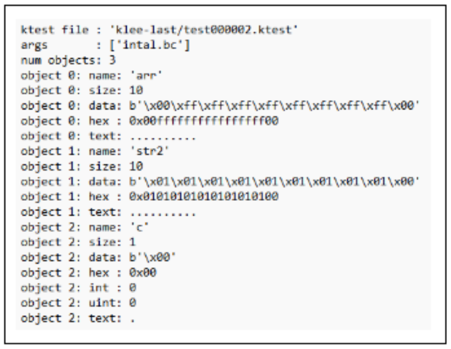
4.2 Performance Analysis

4.2.1 Bugs Found Symbolic Execution of the method append-char() of Intal project generated 2 additional test cases for covering new paths and it detected a memory out of bound pointer error at line 25 of the Intal program (see Fig.12) and it generated a test case that hit the error (see Fig.13). Similarly, all the methods of the Intal program are executed symbolically and KLEE detects two unique errors which are memory out of bound pointer error and free of allocation error.



```
Error: memory error: out of bound pointer
File: intal.c
Line: 25
assembly.ll line: 1654
State: 2
Stack:
	#000001654 in append_char(str=94619018352864, c=symbolic) at intal.c:25
	#100002948 in __user_main(argc=1, argv=94619005985408) at intal.c:561
	#200000449 in __uClibc_main(94619006013848, 1, 94619005985408, 0, 0, 0, 0) at libc/misc/internals/__uClibc_main.c:401
	#300000621 in main(1, 94619005985408)
Info:
	address: 94619018352874
	next: object at 94618994778368 of size 254
		MO22[254] allocated at main():  call void @__uClibc_main(i32 (i32, i8**, i8**)* bitcast (i32 (i32, i8**)* @__user_main
** %1, void ()* null, void ()* null, void ()* null, i8* null)
```

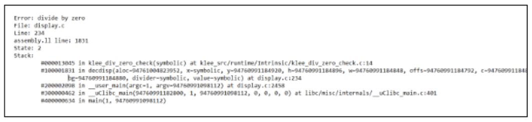
Fig. 12. Memory error detected at line 25



```
ktest file : 'klee-last/test000002.ktest'
args       : ['intal.bc']
num objects: 3
object 0: name: 'arr'
object 0: size: 10
object 0: data: b'\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00'
object 0: hex : 0x00ffffffffffffffff00
object 0: text: ..........
object 1: name: 'str2'
object 1: size: 10
object 1: data: b'\x01\x01\x01\x01\x01\x01\x01\x01\x01\x00'
object 1: hex : 0x01010101010101010100
object 1: text: ..........
object 2: name: 'c'
object 2: size: 1
object 2: data: b'\x00'
object 2: hex : 0x00
object 2: int : 0
object 2: uint: 0
object 2: text: .
```

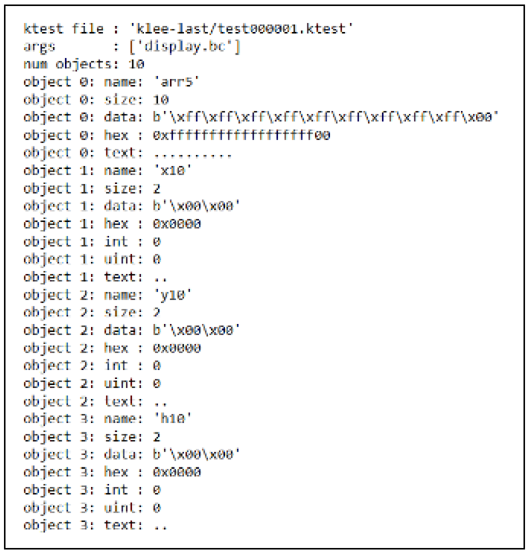
Fig. 13. Test case that hits error at line 25

When the decdisp() method of display.c program was executed symbolically, klee detected 3 errors: divide by zero error at line number 234(Fig.14) and 235 and a memory out of bound error at line number 235. KLEE generated 6 test cases and all of these test cases are executed partially due to the errors. Similarly all the methods in the program characters to Linux buffer the hard way has been executed symbolically and KLEE detected two different errors for this program: divide by zero error and memory out of bound pointer error. See Fig.15 for the test case that threw divide by zero error at line number 234.



```
Error: divide by zero
File: display.c
Line: 234
assembly.ll line: 1831
State: 2
Stack:
        #000013045 in klee_div_zero_check(symbolic) at klee_src/runtime/Intrinsic/klee_div_zero_check.c:14
        #100001831 in decdisp(aloc=94761004823952, x=symbolic, y=94760991184920, h=94760991184896, w=94760991184848, offs=94760991184792, c=94760991184
                bg=94760991184880, divider=symbolic, value=symbolic) at display.c:234
        #200002098 in __user_main(argc=1, argv=94760991098112) at display.c:2458
        #300000462 in __uClibc_main(94760991182800, 1, 94760991098112, 0, 0, 0, 0) at libc/misc/internals/__uClibc_main.c:401
        #400000634 in main(1, 94760991098112)
```

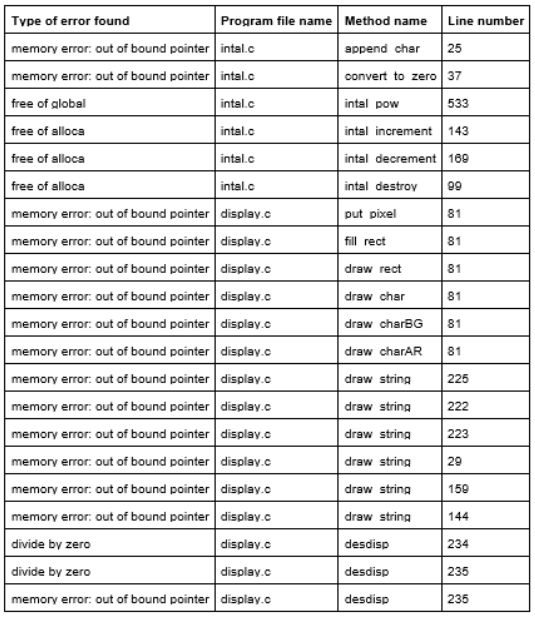
Fig. 14. Divide by zero error detected at line number 234



```
ktest file : 'klee-last/test000001.ktest'
args       : ['display.bc']
num objects: 10
object 0: name: 'arr5'
object 0: size: 10
object 0: data: b'\xff\xff\xff\xff\xff\xff\xff\xff\xff\x00'
object 0: hex : 0xffffffffffffffffff00
object 0: text: ..........
object 1: name: 'x10'
object 1: size: 2
object 1: data: b'\x00\x00'
object 1: hex : 0x0000
object 1: int : 0
object 1: uint: 0
object 1: text: ..
object 2: name: 'y10'
object 2: size: 2
object 2: data: b'\x00\x00'
object 2: hex : 0x0000
object 2: int : 0
object 2: uint: 0
object 2: text: ..
object 3: name: 'h10'
object 3: size: 2
object 3: data: b'\x00\x00'
object 3: hex : 0x0000
object 3: int : 0
object 3: uint: 0
object 3: text: ..
```

Fig. 15. Test case that threw the error at line 234

Details of errors detected by KLEE is illustrated in Fig.16



Type of error found	Program file name	Method name	Line number
memory error: out of bound pointer	intal.c	append char	25
memory error: out of bound pointer	intal.c	convert to zero	37
free of global	intal.c	intal pow	533
free of alloca	intal.c	intal increment	143
free of alloca	intal.c	intal decrement	169
free of alloca	intal.c	intal destroy	99
memory error: out of bound pointer	display.c	put pixel	81
memory error: out of bound pointer	display.c	fill rect	81
memory error: out of bound pointer	display.c	draw rect	81
memory error: out of bound pointer	display.c	draw char	81
memory error: out of bound pointer	display.c	draw charBG	81
memory error: out of bound pointer	display.c	draw charAR	81
memory error: out of bound pointer	display.c	draw string	225
memory error: out of bound pointer	display.c	draw string	222
memory error: out of bound pointer	display.c	draw string	223
memory error: out of bound pointer	display.c	draw string	29
memory error: out of bound pointer	display.c	draw string	159
memory error: out of bound pointer	display.c	draw string	144
divide by zero	display.c	desdisp	234
divide by zero	display.c	desdisp	235
memory error: out of bound pointer	display.c	desdisp	235

Fig. 16. List of errors found

It is observed that KLEE is able to complete the execution and generate test cases for discovered paths for some of the methods based on the size and complexity of methods. On the other hand, KLEE takes a higher amount of time for execution if the method is larger and also for methods dealing with memory read and write operations. This results in a higher waiting period for viewing the KLEE output is observed as one of the limitations of KLEE Dynamic Symbolic Execution. Flags such as -max-memory and -max-time can be used for making KLEE to halt the execution and produce results generated by specifying the maximum memory usage and time duration respectively. This will again reduce the effectiveness of KLEE and will not be able to cover all the execution paths and restricts KLEE in generating useful test cases. KLEE execution also depends on the number and type of input parameters to a method.

5 Conclusion

In this project, we performed a study of KLEE Dynamic Symbolic Execution results of three open source projects and the effectiveness of the system in generating test cases. The results show that KLEE automatically generated test cases that covered program execution paths and detected different types of errors.

KLEE based test suites are able to achieve higher coverage but time required to complete the execution depends on many factors such as method size, number and types of parameters and complexity of computations involved. Overall, KLEE is an effective Symbolic Execution tool for achieving a significant amount of coverage and it provides useful techniques for uncovering program errors.

References

1. Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In 33rd International Conference on Software Engineering (ICSE). IEEE, 1066–1071.
2. Zhang T., Wang P., Guo X. A survey of symbolic execution and its tool KLEE. Procedia Computer Science,2020
3. Roberto Baldoni, Emilio Coppa, Daniele C. D'elia, Camil Demetrescu, and Irene Finocchi. 2019. A survey of symbolic execution techniques. ACM Computing Surveys, 51, 3 (May 2019)
4. Integers of Arbitrary Length open source Git project, https://github.com/Aveek-Saha/Intal
5. Docker.KLEE, http://klee.github.io/docker/
6. Software testing: fuzzing and concolic execution with AFL, KLEE and ANGR, https://www.benhup.com/security-privacy/software-testing-fuzzing-concolic-execution-with-afl-klee-and-angr/
7. Tiny-Regex open source Git project, https://github.com/kokke/tiny-regex-c
8. Characters To Linux Buffer open source Git project, https://github.com/torvalds/linux/blob/master/fs/buffer.c