



## **ECE-653 SOFTWARE TESTING, QUALITY ASSURANCE AND MAINTENANCE**

UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

---

# **Symbolic execution using KLEE**

---

*Authors:*

Sheena Agrawal (ID: 20949433)

Arshad Momin (ID: 20986176)

Karthikeyan Subramanian (ID: 20987035)

Date: August 11, 2022

## Contents

<b>1</b>	<b>Abstract</b>	<b>3</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
2.1	Symbolic Execution . . . . .	3
2.2	Dynamic Symbolic Execution . . . . .	4
2.3	KLEE . . . . .	4
2.4	Lexical analyzer . . . . .	4
2.5	Tic-Tac-Toe . . . . .	5
<b>3</b>	<b>Implementation</b>	<b>5</b>
3.1	Setting up KLEE . . . . .	5
3.2	Making the code base KLEE compatible . . . . .	5
3.2.1	Lexical analyzer . . . . .	6
3.2.2	Tic-Tac-Toe . . . . .	6
3.3	Running KLEE . . . . .	7
3.3.1	Lexical analyzer . . . . .	7
3.3.2	Tic-Tac-Toe . . . . .	8
<b>4</b>	<b>Implementation-Results</b>	<b>8</b>
4.1	Lexical analyzer - Results . . . . .	8
4.2	Tic-Tac-Toe - Results . . . . .	11
<b>5</b>	<b>Limitation</b>	<b>13</b>
<b>6</b>	<b>Conclusion</b>	<b>14</b>
<b>7</b>	<b>References</b>	<b>15</b>

## 1 Abstract

Dynamic symbolic execution is an effective testing technique which generates test cases automatically. These test cases generate inputs which trigger bugs in a piece of software. These bugs could be low level bugs that trigger program crashes or even high level bugs caused by semantic properties. Dynamic symbolic execution is extremely effective to achieve high coverage of code and also provides per-path correctness guarantees. The idea of this project is to leverage the properties of symbolic execution using the tool KLEE on a lexical analyzer. This project intends to check the correctness of the program, generate test cases and consecutively detect bugs. This project will also report the code coverage.

## 2 Introduction

### 2.1 Symbolic Execution

Symbolic execution is a software testing approach that can help with test data creation and confirming program quality. The execution requires a collection of data values to exercise a set of pathways.

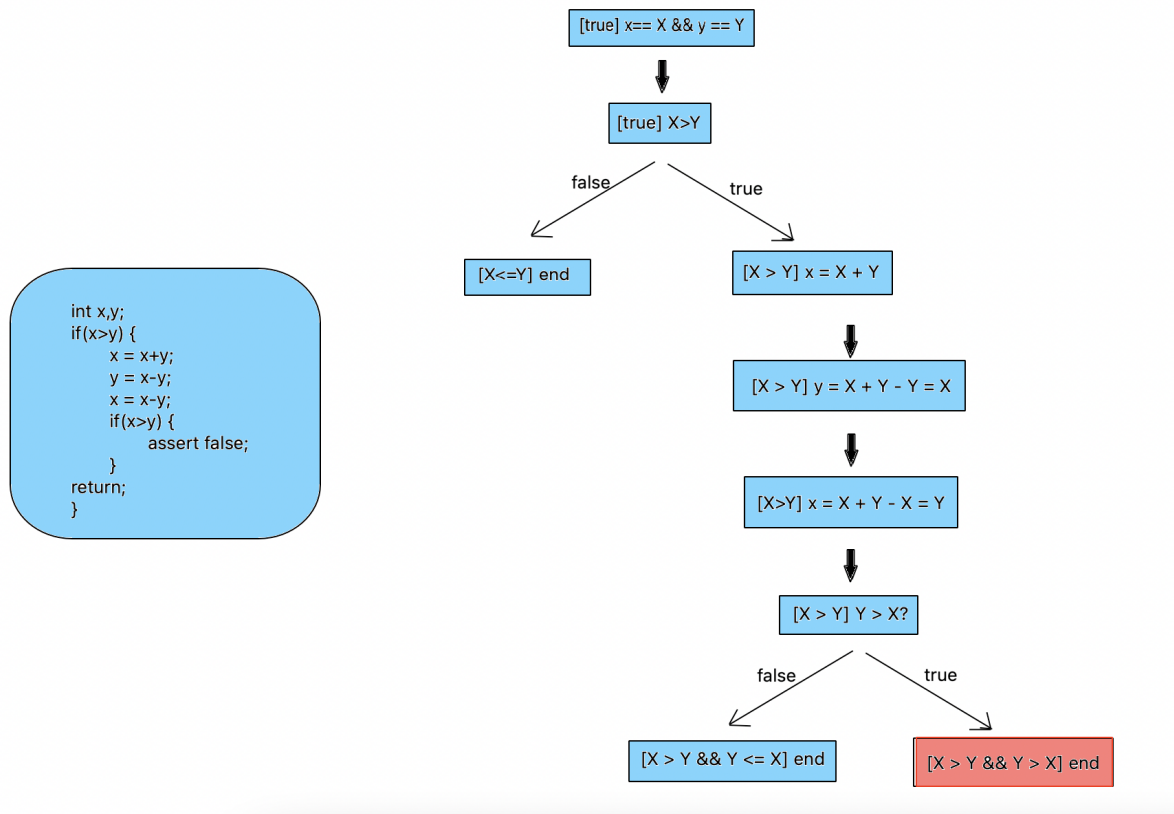


Fig 1. Symbolic Execution

The data is replaced with symbolic values with a series of expressions in symbolic execution, one expression for each output variable. The most frequent strategy for symbolic execution is to analyse the program, which results in the production of a flow graph. The flow graph identifies the decision points and assigned tasks for each flow. A set of assignment statements and branch predicates is generated by traversing the flow graph from an entry point.

## 2.2 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) is a path-based symbolic execution technique founded on two principles. To begin, the technique runs program P on some input seeding the symbolic execution process with a viable path. Second, where symbolic reasoning is neither possible or desirable, DSE substitutes concrete values from the execution of the input seed for symbolic formulations. The primary advantage of DSE is that it simplifies the development of a symbolic execution tool by leveraging concrete execution behaviour (by actually running the program). Because DSE includes both concrete and symbolic logic, it has also been referred to as "concolic" execution.

## 2.3 KLEE

KLEE is a dynamic symbolic execution tool capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs[1]. This project utilized KLEE image built locally and installed all required softwares in the KLEE container locally. The project describes in detail all the steps followed to run KLEE successfully.

## 2.4 Lexical analyzer

Lexical analysis is the initial phase of the compiler, sometimes known as a scanner. It turns the High level input program into a Token sequence. A lexical token is a sequence of letters that can be considered as a unit in programming language grammar. A lexeme is a series of characters that are matched by a pattern to generate the associated token or a sequence of input characters that compose a single token. For example, "int", "sample variable", "+", "=", "987", ":"

This Lexical Analyzer software converts a stream of single characters, usually organised as lines, into a stream of lexical tokens. The main goal/purpose of the project is to take a C file and create a series of tokens that may be used in the next stage of compilation. This should also take into consideration any error handling needs that may emerge during tokenization.

## 2.5 Tic-Tac-Toe

In the two-player board game of tic tac toe, commonly known as noughts and crosses or Xs and Os, players alternately mark squares in a three-by-three grid with a 'X' or an 'O'. The person who correctly arranges three of their markers in a line, whether it be vertical, horizontal, or diagonal, wins. Using 1D arrays and the C programming language, you can create this entertaining game. When writing a Tic Tac Toe game in the C programming language, arrays are crucial.

To keep track of the game's development, the Xs and Os are saved in distinct arrays and passed between various methods in the code. Entering the code with provision to choose either X or O to play the game.

## 3 Implementation

### 3.1 Setting up KLEE

To set up KLEE, below steps were followed:

#### Building the image locally:

```
git clone https://github.com/klee/klee.git
cd klee
docker build -t klee/klee .
```

#### Creating a local KLEE container:

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

#### Installing Git in the container:

```
(Use sudo in case you have any permission issues)
apt-get update
apt-get upgrade -y
apt-get install -y git
```

#### Installing clang in the container:

```
(Use sudo in case you have any permission issues)
apt update
apt install -y clang
```

### 3.2 Making the code base KLEE compatible

We choose 2 different softwares, first one is Lexical analyzer and second is the popular tic-tac-toe game. The code for the both were referenced from GitHub and were modified to become compatible with KLEE.

### 3.2.1 Lexical analyzer

While lexical analysis requires individual characters as inputs to the system, for symbolic analysis the input provided to KLEE was an array with a sequence of characters. The char " " acts as terminator for the sequence of inputs. The max array size was set to be 4 bytes so as to limit the memory that test cases require to run.

```

please input string:
a d d 1 2 3 4 5 > < = #
(10, a)
(10, d)
(10, d)
(11, 1)
(11, 2)
(11, 3)
(11, 4)
(11, 5)
(23, >)
(20, <)
input error
( 0, #)
sh: 1: pause: not found
klee@6910257be076:~/project-k3subram-a5momin-s37agraw$

```

Fig 2. Lexical analyzer Without Klee.

### 3.2.2 Tic-Tac-Toe

The popular game is a 2-player game that requires players to alternatively input 'X' or 'O' as inputs. The code base has been modified to accept an array of 9 bytes. The program will terminate once either player wins the game or if there is a draw. There are 2 restrictions applied to the array. This array will only accept 'X' and 'O' as inputs. Also, the number of 'X' should be only 1 greater than 'O', or vice versa. These restrictions are applied to simulate an actual game being played between 2 players.

```

X   |   |   |
-----|-----|-----
O   |   |   |
-----|-----|-----
X   |   |   |
-----|-----|-----
X WON
klee@b3d67fd2561b:~/chrome-dinosaur$

```

Fig 3. Tic-Tac-Toe without Klee.

### 3.3 Running KLEE

We first need to compile the code in LLVM bit code format. The below command is used:

```
clang -c -g -emit-llvm {filename.c}
```

To run KLEE, the below command is used:

```
klee {filename.bc}
```

To run KLEE more efficiently, we used the below commands in addition for running KLEE:

```
--libc=uclibc : This command is used for added support for string functions
--posix-runtime : enables the use of the symbolic environment options
                  as part of the program's options
--only-output-states-covering-new : limits the test generation to states
                                   that actually covered new code
--max-tests=50 : limits the number of maximum test cases in a particular
                 run to 50
--external-calls=all :
--max-time=300 :limits the time in a particular run to 5 mins
--write-cov : writes coverage of the test cases covered
--write-test-info : records additional information about the test cases
--write-paths : records information about the path taken whilst running the
               test cases
--optimize : run the LLVM optimization passes on the bitcode module before
             executing it to remove any dead code
--external-calls=all : allows external calls with concrete arguments
```

#### 3.3.1 Lexical analyzer

Finally, the command used when running KLEE for Lexical analyzer:

```
clang -c -g -emit-llvm klee-lexical-analyzer.c
```

```
klee --libc=uclibc --posix-runtime --external-calls=all --optimize
--only-output-states-covering-new --max-tests=50 --max-time=300
--write-cov --write-test-info --write-paths klee-lexical-analyze.bc
```

### 3.3.2 Tic-Tac-Toe

And the command used when running KLEE for tic-tac-toe:

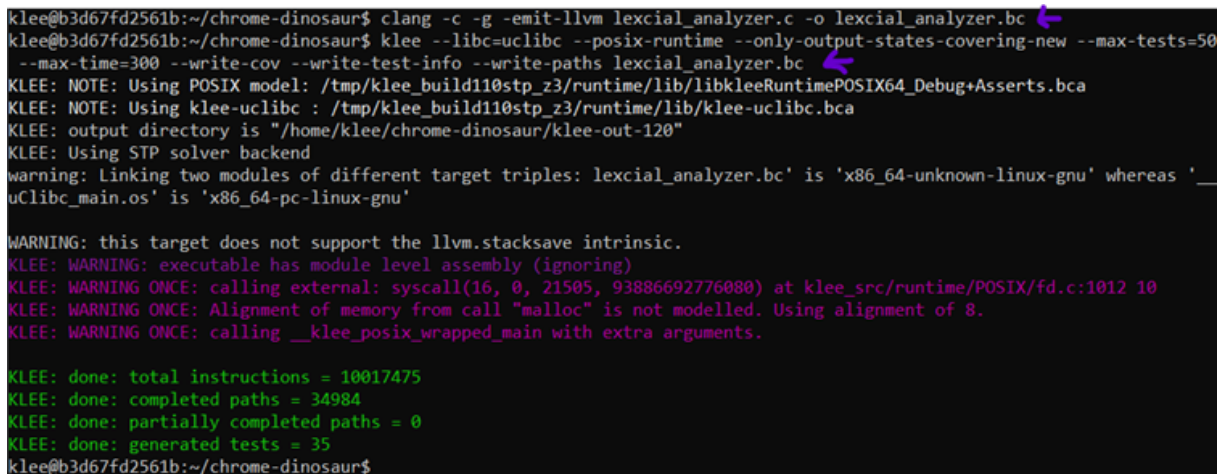
```
clang -c -g -emit-llvm klee-tic-tac-toe.c

klee --libc=uclibc --posix-runtime --external-calls=all --optimize
--only-output-states-covering-new --max-tests=50 --max-time=300
--write-cov --write-test-info --write-paths klee-tic-tac-toe.bc
```

## 4 Implementation-Results

### 4.1 Lexical analyzer - Results

With the implementation of KLEE Symbolic Execution on Lexical Analyzer we observed total instructions, Completed paths, partially generated paths, Number of generated test cases. The below image shows the details of the parameters obtained from KLEE execution



```
klee@b3d67fd2561b:~/chrome-dinosaur$ clang -c -g -emit-llvm lexcial_analyzer.c -o lexcial_analyzer.bc
klee@b3d67fd2561b:~/chrome-dinosaur$ klee --libc=uclibc --posix-runtime --only-output-states-covering-new --max-tests=50
--max-time=300 --write-cov --write-test-info --write-paths lexcial_analyzer.bc
KLEE: NOTE: Using POSIX model: /tmp/klee_build110stp_z3/runtime/lib/libkleeRuntimePOSIX64_Debug+Asserts.bca
KLEE: NOTE: Using klee-uclibc : /tmp/klee_build110stp_z3/runtime/lib/klee-uclibc.bca
KLEE: output directory is "/home/klee/chrome-dinosaur/klee-out-120"
KLEE: Using STP solver backend
warning: Linking two modules of different target triples: lexcial_analyzer.bc' is 'x86_64-unknown-linux-gnu' whereas '
uClibc_main.os' is 'x86_64-pc-linux-gnu'

WARNING: this target does not support the llvm.stacksave intrinsic.
KLEE: WARNING: executable has module level assembly (ignoring)
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 93886692776080) at klee_src/runtime/POSIX/fd.c:1012 10
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling __klee_posix_wrapped_main with extra arguments.

KLEE: done: total instructions = 10017475
KLEE: done: completed paths = 34984
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 35
klee@b3d67fd2561b:~/chrome-dinosaur$
```

Fig.4 Klee Results for Lexical analyzer

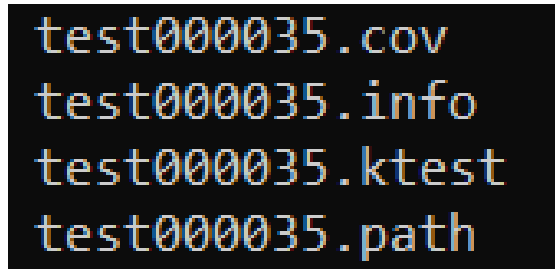
To iterate, below commands were used to generate .cov, .info, .path and .ktest files.

```
clang -c -g -emit-llvm lexical-analyzer.c

klee --libc=uclibc --posix-runtime --external-calls=all --optimize
--only-output-states-covering-new --max-tests=50 --max-time=300
--write-cov --write-test-info --write-paths lexcial_analyzer.bc
```



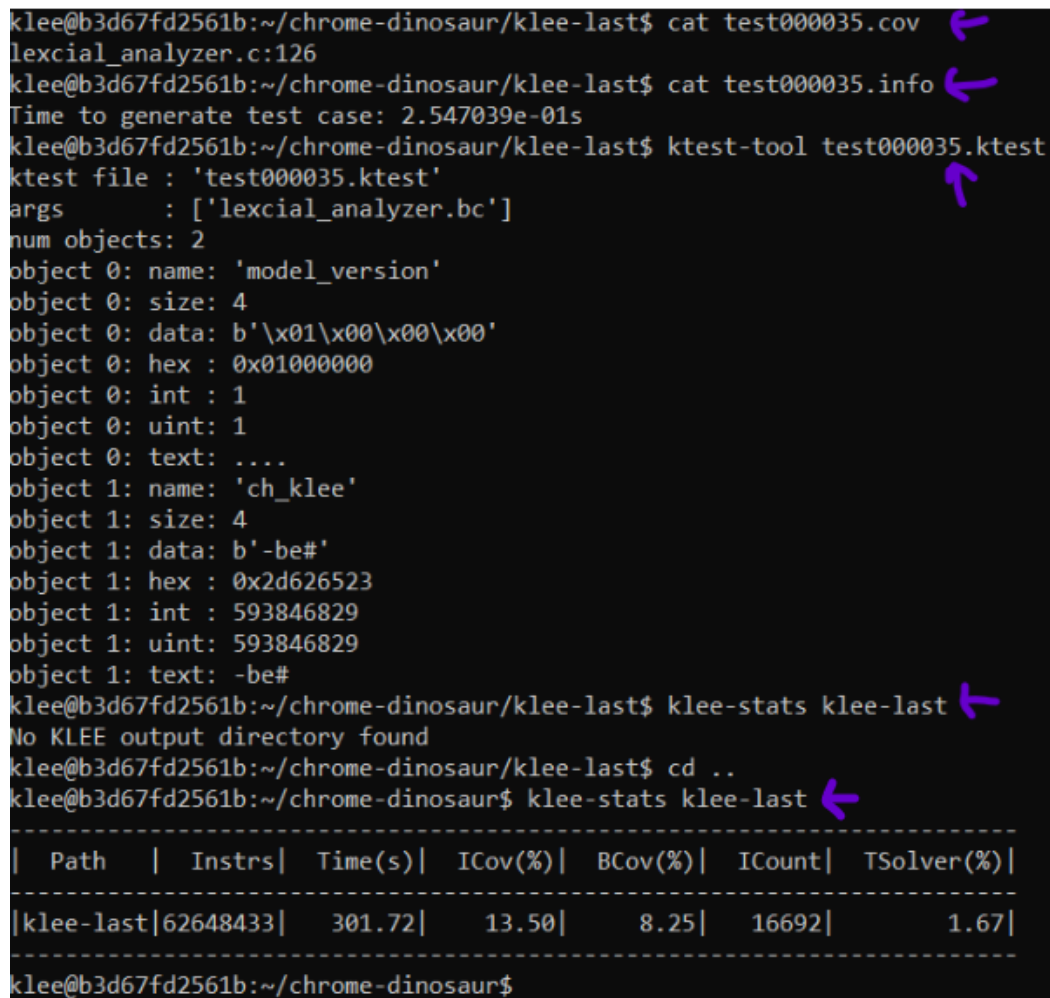
Using the commands mentioned above the following type of files are generated for each test case:



```
test000035.cov
test000035.info
test000035.ktest
test000035.path
```

Fig.5 Files Generated with Klee Execution.

Below image contains the .cov, .info, .ktest files and its detailed information:



```
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ cat test000035.cov
lexcial_analyzer.c:126
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ cat test000035.info
Time to generate test case: 2.547039e-01s
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ ktest-tool test000035.ktest
ktest file : 'test000035.ktest'
args      : ['lexcial_analyzer.bc']
num objects: 2
object 0: name: 'model_version'
object 0: size: 4
object 0: data: b'\x01\x00\x00\x00'
object 0: hex : 0x01000000
object 0: int : 1
object 0: uint: 1
object 0: text: ....
object 1: name: 'ch_klee'
object 1: size: 4
object 1: data: b'-be#'
object 1: hex : 0x2d626523
object 1: int : 593846829
object 1: uint: 593846829
object 1: text: -be#
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ klee-stats klee-last
No KLEE output directory found
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ cd ..
klee@b3d67fd2561b:~/chrome-dinosaur$ klee-stats klee-last
```

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	62648433	301.72	13.50	8.25	16692	1.67

```
klee@b3d67fd2561b:~/chrome-dinosaur$
```

Fig.6 Information about Test Case.

We even discovered a few bugs and re-ran KLEE after fixing them up. Below is the screenshot of a sample bug found in Lexical analyzer:

```
klee@b3d67fd2561b:~/chrome-dinosaur$ clang -g -c -emit-llvm lexcial_analyzer.c
klee@b3d67fd2561b:~/chrome-dinosaur$ klee --libc=uclibc --posix-runtime --external-calls=all lexcial_analyzer.bc
KLEE: NOTE: Using POSIX model: /tmp/klee_build110stp_z3/runtime/lib/libkleeRuntimePOSIX64_Debug+Asserts.bca
KLEE: NOTE: Using klee-uclibc : /tmp/klee_build110stp_z3/runtime/lib/klee-uclibc.bca
KLEE: output directory is "/home/klee/chrome-dinosaur/klee-out-131"
KLEE: Using STP solver backend
warning: Linking two modules of different target triples: lexcial_analyzer.bc' is 'x86_64-unknown-linux-gnu' whereas
'__uClibc_main.os' is 'x86_64-pc-linux-gnu'

WARNING: this target does not support the llvm.stacksave intrinsic.
KLEE: WARNING: executable has module level assembly (ignoring)
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 94375072537776) at klee_src/runtime/POSIX/fd.c:1012 10
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling __klee_posix_wrapped_main with extra arguments.
KLEE: WARNING ONCE: skipping fork (memory cap exceeded)
KLEE: WARNING: killing 11996 states (over memory cap: 2101MB)
KLEE: ERROR: lexcial_analyzer.c:49: memory error: out of bound pointer ←
KLEE: NOTE: now ignoring this error at this location
^CKLEE: ctrl-c detected, requesting interpreter to halt.
KLEE: halting execution, dumping remaining states

KLEE: done: total instructions = 24936395
KLEE: done: completed paths = 3078
KLEE: done: partially completed paths = 248192
KLEE: done: generated tests = 251267
klee@b3d67fd2561b:~/chrome-dinosaur$
```

Fig.7a Lexical analyzer Bug - memory error: out of bound pointer

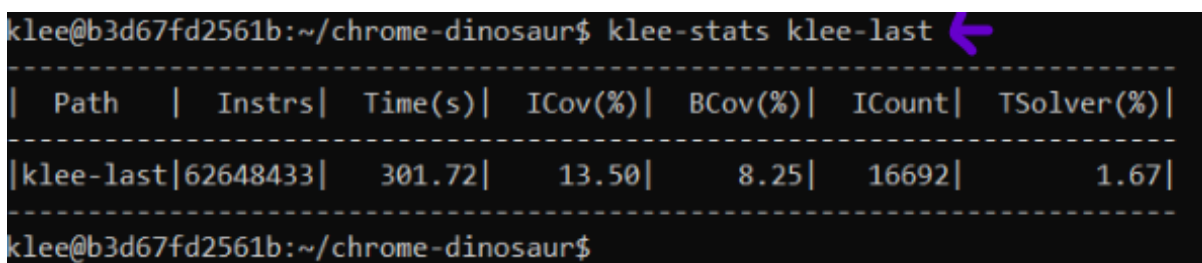
```
klee@b3d67fd2561b:~/chrome-dinosaur$ klee --libc=uclibc --posix-runtime --only-output-states-covering-new --max-t
ests=50 --max-time=300 --write-cov --write-test-info --write-paths klee-lexical-analyzer.bc
KLEE: NOTE: Using POSIX model: /tmp/klee_build110stp_z3/runtime/lib/libkleeRuntimePOSIX64_Debug+Asserts.bca
KLEE: NOTE: Using klee-uclibc : /tmp/klee_build110stp_z3/runtime/lib/klee-uclibc.bca
KLEE: output directory is "/home/klee/chrome-dinosaur/klee-out-124"
KLEE: Using STP solver backend
warning: Linking two modules of different target triples: klee-lexical-analyzer.bc' is 'x86_64-unknown-linux-gnu'
whereas '__uClibc_main.os' is 'x86_64-pc-linux-gnu'

WARNING: this target does not support the llvm.stacksave intrinsic.
KLEE: WARNING: executable has module level assembly (ignoring)
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 94150691314960) at klee_src/runtime/POSIX/fd.c:1012 1
0
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled. Using alignment of 8.
KLEE: WARNING ONCE: calling __klee_posix_wrapped_main with extra arguments.
KLEE: ERROR: lexcial_analyzer.c:151: free of global
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 7148862
KLEE: done: completed paths = 0
KLEE: done: partially completed paths = 34984
KLEE: done: generated tests = 1
klee@b3d67fd2561b:~/chrome-dinosaur$
```

Fig.7b Lexical analyzer Bug - free of global

Below image shows how to get details by using "Klee-stats" command:



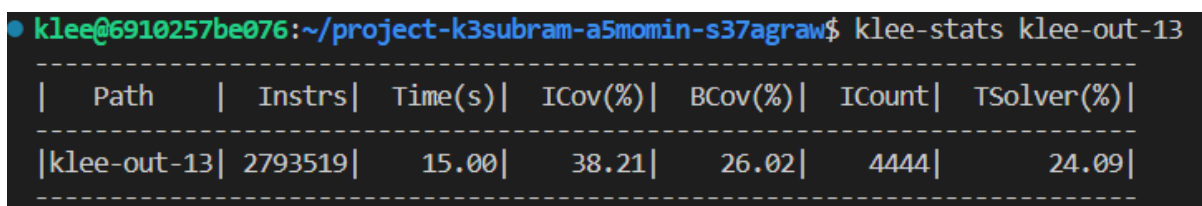
```
klee@b3d67fd2561b:~/chrome-dinosaur$ klee-stats klee-last
```

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	62648433	301.72	13.50	8.25	16692	1.67

```
klee@b3d67fd2561b:~/chrome-dinosaur$
```

Fig.8 File Information

We observed that the ICov(percentage of LLVM instructions) and the BCov(percentage of branches that were covered) was very low, and then we ran the tool again with `--optimize` parameter, and the coverage increased significantly. Below is the result:



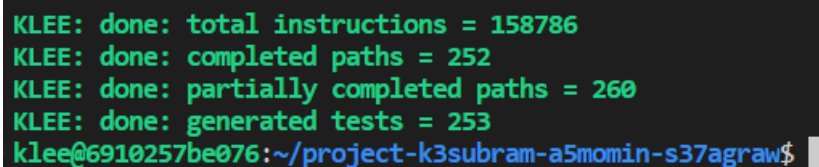
```
klee@6910257be076:~/project-k3subram-a5momin-s37agraw$ klee-stats klee-out-13
```

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-out-13	2793519	15.00	38.21	26.02	4444	24.09

Fig.9 Lexical analyzer Optimized

## 4.2 Tic-Tac-Toe - Results

After implementing Tic-Tac-Toe, we obtained number of total instructions, completed paths, partially generated paths, number of generated test cases by running KLEE. The below image shows the details of the parameters obtained from KLEE:



```
KLEE: done: total instructions = 158786
KLEE: done: completed paths = 252
KLEE: done: partially completed paths = 260
KLEE: done: generated tests = 253
klee@6910257be076:~/project-k3subram-a5momin-s37agraw$
```

Fig.10 TicTacToe- Klee results

To iterate, in order to get unique test cases from Klee symbolic execution we used the below command:

```
clang -c -g -emit-llvm my-tic-tac-toe-klee.c
```

```
klee --libc=uclibc --posix-runtime --external-calls=all --only-output-states-
covering-new --max-tests=50 --max-time=300 --write-cov --write-test-info --optimize
--write-paths --optimize my-tic-tac-toe.bc
```

Total combinations of 'X' and 'O' : 2 power 9

The above commands generates the .info, .path, .cov and .ktest files.

```
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ ll test000048.*
-rw-r--r-- 1 klee klee  20 Aug  9 20:04 test000048.cov
-rw-r--r-- 1 klee klee  42 Aug  9 20:04 test000048.info
-rw-r--r-- 1 klee klee  93 Aug  9 20:04 test000048.ktest
-rw-r--r-- 1 klee klee 1840 Aug  9 20:04 test000048.path
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$
```

Fig.11 Generated Files using Klee Command

The below figure shows .cov, .info, .ktest Files and its detailed information

We run kstats command to show klee's processing

```
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ ktest-tool test000048.ktest
ktest file : 'test000048.ktest'
args       : ['my-tic-tac-toe.bc']
num objects: 2
object 0: name: 'model_version'
object 0: size: 4
object 0: data: b'\x01\x00\x00\x00'
object 0: hex : 0x01000000
object 0: int : 1
object 0: uint: 1
object 0: text: ....
object 1: name: 'board'
object 1: size: 9
object 1: data: b'OXOXOXOX'
object 1: hex : 0x4f584f4f584f584f58
object 1: text: OXOXOXOX
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ cat test000048.cov
my-tic-tac-toe.c:60
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$ cat test000048.info
Time to generate test case: 6.158000e-03s
klee@b3d67fd2561b:~/chrome-dinosaur/klee-last$
```

Fig.12 Test Case, Coverage Info and Time to generate test case

Below image shows how to get details by using "Klee-stats" command:

```

KLEE: done: total instructions = 260571
KLEE: done: completed paths = 512
KLEE: done: partially completed paths = 0
KLEE: done: generated tests = 48
klee@b3d67fd2561b:~/chrome-dinosaur$ klee-stats klee-last
-----
| Path      | Instrs | Time(s) | ICov(%) | BCov(%) | ICount | TSolver(%) |
-----
| klee-last | 260571 | 1.14    | 11.85   | 11.47   | 17010  | 27.44      |
-----
klee@b3d67fd2561b:~/chrome-dinosaur$

```

Fig.13 Klee-Stats Details (Paths, Instructions, Time, Coverage, etc)

We observed that the ICov(percentage of LLVM instructions) and the BCov(percentage of branches that were covered) was very low, and then we ran the tool again with `-optimize` parameter, and the coverage increased significantly. Below is the result:

```

klee@b3d67fd2561b:~/chrome-dinosaur$ klee-stats klee-out-135
-----
| Path      | Instrs | Time(s) | ICov(%) | BCov(%) | ICount | TSolver(%) |
-----
| klee-out-135 | 258102 | 0.99    | 38.32   | 37.37   | 4449   | 27.13      |
-----
klee@b3d67fd2561b:~/chrome-dinosaur$

```

Fig.14 Tic-Tac-Toe Optimized

## 5 Limitation

1. The memory model utilised by KLEE's LLVM IR interpreter is memory and time expensive. KLEE has a separate "address space" for each execution route. There is a "stack" for local variables and a "heap" for global variables and dynamically allocated variables in the address space. Each variable (local or global) is wrapped in a memory object that maintains metadata related to this variable, such as the starting address, size, and allocation information. Each variable's memory size would equal the size of the original variable plus the size of the memory object. When attempting to access a variable, KLEE first searches the "address space" for the associated memory object. KLEE would examine this memory object to determine the authenticity of the access. If this is the case, the access will be performed and the memory object's state will be updated. This type of memory access is clearly slower than RAM. A design like this may

readily enable the spread of symbolic values.

2. Models for system environments are lacking in KLEE. A basic file system is the sole system component modelled in KLEE. Sockets and multi-threading, for example, are not supported. When a program contacts certain unmodified components, KLEE either terminates the execution and issues an alarm, or redirects the call to the underlying OS.

3. KLEE cannot deal directly with binaries. KLEE requires the LLVM IR of an untested program. Other BitBlaze Symbolic Execution tools, such as S2E and VINE, may deal with binaries.

## 6 Conclusion

Performing Symbolic execution using Klee we were able to detect few bugs in the Code. The rectification of the bug makes it clear for the Klee implementation to generate the test cases and provide details such as Completed paths, Partially completed paths, Total number of Instructions and Number of Generated Test cases. These parameters were observed for Lexical Analyzer and Tic-Tac-Toe Game. The table below shows a tabular view of the results achieved by running symbolic execution using KLEE:

	Lexical Analysis	Tic Tac Toe
<b>Time (s)</b>	301.72	1.14
<b>Total path</b>	34984	512
<b>Bugs detected</b>	2	0
<b>Total test cases</b>	35	48
<b>Total instructions</b>	10017475	26057
<b>BCov</b>	26.02%	37.37%
<b>ICov</b>	38.21%	38.32%
<b>T-Solver</b>	24.09	27.13

Fig.15 Conclusion Table

## 7 References

1. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (<https://llvm.org/pubs/2008-12-OSDI-KLEE.html>)
2. Deconstructing Dynamic Symbolic Execution (<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/dse.pdf>)
3. Lexical analyzer sample code in C (<https://gist.github.com/luckyshq/6749155>)
4. Tic-Tac-Toe sample code in C (<https://github.com/thinkphp/tic-tac-toe/blob/master/tic-tac-toe.c>)
5. Getting Started with KLEE (<https://klee.github.io/>)
6. Overview of KLEE's main command-line options (<https://klee.github.io/docs/options/>)
7. Symbolic execution with KLEE (<https://adalogics.com/blog/symbolic-execution-with-klee>)