KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler
Stanford University
Presented by Adam Bergstein
November 28, 2011

Outline

- Background
  - Symbolic execution
  - Constraints and solvers
  - Sinks/sink sources
  - Abstract domain and concretization
  - System modeling
- KLEE
  - Main concepts
  - Overall process
  - Precision from LLVM and bytecode
  - Notion of states
  - Constraints and paths
  - Performance and Environment
  - Results
- My Thoughts
- Questions

Background

- Symbolic execution
  - Simulation that approximates variable values by using symbols
  - Operations on variables constrain the symbols
  - Used to reason about possible values that cause certain conditions in a program
    - Is a symbolic value in the range of values that cause something to occur?
  - http://www.stat.uga.edu/stat_files/billard/tr_symbolic.pdf
- Constraints and solvers
  - Constraints are collected facts about a program that define bounds on possible execution at specific points in a program
  - Solvers determine the possibility of concrete values based on the constraints
  - Certain concrete values can conditionally cause programs to behave in undesirable ways

Background

- Sinks and sink sources
  - Sinks identify meaningful operations within the code
  - Sources identify the data origins that can influence sinks
- Abstract domain and concretization
  - Defining the range of all possible values for variables
  - Concretization maps actual variable values from ranges of possible values
- System modeling
  - "Approximating" how a system behaves when it runs
  - We have looked at different ways to represent systems, like CFGs, summary functions, etc

KLEE > Main Concepts

- Use of static analysis to determine if there are possible concrete values that cause vulnerabilities in the program
- Simulate a program and leverage symbolic execution
- Build constraints and maintain a series of states throughout the simulation
  - States define each unique path throughout the program
- Leverage a solver to determine possibilities within the program based on constraints
  - Return concrete values if something was solvable
- Document areas of the code that have any possible values that can cause vulnerabilities
  - Based on a set of possible dangerous operations
- "Based on the ***constraints*** (state of unique path) at the time I get to this line of code with a potentially dangerous operation, is there ***any possible value*** that can cause this line of code to be ***dangerous***?"

KLEE > Main Concepts

- KLEE begins by constructing unconstrained variables for arguments into state
  - Initial constraints are set based on *--sym-args* when running KLEE
  - Defines number of arguments and number of characters per argument
  - Sets initial constraints so operation is not totally unbounded
- Analysis simulates each instruction and runs each state per instruction
  - Scheduling algorithm to select which state to analyze first
  - Collect more constraints, update the symbolic values in the state
  - When reaching a potential operation that contains an exit or error, look at the ***path condition***
- Path conditions are the collection of constraints that are valid for that specific path
  - A path condition is unique for each state since a path can influence the symbolic values on a path by path basis
  - On a branch statement, a state is cloned for possible paths
  - The path condition is updated per state, to mimic unique paths
- Determining malicious concrete values are bounded by the path condition
  - These are sent to STP solver
  - Is there a possible set of values that can cause an issue?

KLEE > Overall Process

- Compile program into bytecode with LLVM
- Run KLEE with defined number of arguments and initial character bound constraints of arguments
  - Assists with abstract domain to make it bounded
- Simulate the program, symbolic execution
  - Collect constraints on variables, update state
- For branches, determine what is possible based on constraints
  - Pass constraints to solver to see what branch is possible
  - Clone state for all possible branches, update path conditions in each state
  - Similar to may/must analysis
- For potential dangerous operations, identify any concrete values that cause dangerous operations
  - Pass constraints to solver
  - Return any possible values that can cause undesired results
- Useful for bounds checking, pointer dereferencing, assertions

KLEE > Precision from LLVM byte code

- The constraints are very precise because the byte code represents bit-level accuracy
- This reduces the approximation used in modeling the running application
- This precision makes the solver more effective in determining possible values

KLEE > Notion of States

- Each state represents one unique path in the program at a given point in runtime
- Need to maintain symbolic values by state at the given instruction
- Maintains register file, stack, heap, program counter
  - Instruction pointer is maintained by KLEE
- Maintain constraints of the path conditions for use within the solver
  - States may be active or inactive for a given instruction based on path condition and constraints

KLEE > Constraints and Paths

- The goal is to find concrete values that cause dangerous operations
- For the solver to be effective in finding concrete values, the abstract domain needs to be reduced
- Path conditions set constraints on variable values of the specific path
  - $i<0$, $j==10$, etc
- Symbolic values creates its own constraints on variables
  - $i = (2 \times i) + 10$
  - $j = j^2$
- The combination of symbolic values and path conditions set bounds for the solver to determine possible values based on state for a given instruction

KLEE > Performance and Environment

- Two of the biggest challenges were performance and modeling operations involving the environment
- The number of states can grow rapidly
  - To combat it, KLEE uses a shared memory mapping between states
- Use of compiler-like tricks to make problems easier for the solver
- Environment calls are modeled by C code, to reflect the runtime state
  - Use of uClibc to mimic system calls
  - KLEE developers have set up other custom models to reflect operations involving the environment

KLEE > Results

- Looked at packages which supported common command-line programs like *ls* and *tr*
- Average of 90% code coverage
- Highlighted differences between in CoreUtils and Busybox
  - Simulated the same commands and found differences between the two packages
- Found errors in both CoreUtils and Busybox, respectively

	Coreutils		Busybox	
Coverage (w/o lib)	**KLEE tests**	**Devel. tests**	**KLEE tests**	**Devel. tests**
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

```
t1.txt: "\t \tMD5("
t2.txt: "\b\b\b\b\b\b\b\t"
t3.txt: "\n"
t4.txt: "a"
```

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

```
date -I
ls --co
chown a.a -
kill -l a
setuidgid a ""
printf "% *" B
od t1.txt
od t2.txt
printf %
printf %Lo
tr [
tr [=
tr [a-z
```

```
t1.txt: a
t2.txt: A
t3.txt: \t\n
```

```
cut -f t3.txt
install --m
nmeter -
envdir
setuidgid
envuidgid
envdir -
arp -Ainet
tar tf_ /
top d
setarch "" ""
<full-path>/linux32
<full-path>/linux64
hexdump -e ""
ping6 -
```

Figure 10: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in BUSYBOX. When multiple applications crash because of the same shared (buggy) piece of code, we group them by shading.

Differences between CoreUtils and Busybox

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[in□nite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/: Is a directory"`
`split /`	`"/: Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[ 0 ‘‘<’’ 1 ]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

My Thoughts

- There are a lot of similarities from what we have discussed in class
  - PHP paper used sinks and sink sources with query statements
  - This paper looks for operations like pointers, assertions, printf, and load/stores
  - Symbolic execution like the PHP paper
  - May/must analysis for looking at potential paths
  - Constraints and use of a solver
    - Constraints defined by symbolic analysis and paths
  - Can be considered context and flow sensitive
    - Creates new states based on path branches
    - Simulates function calls per state based on the current state values
  - Concretization based on symbolic values and path conditions

My Thoughts

- There are some differences between the approaches
  - No mention of a control flow graph, purely a simulation tool
  - Their goal is only to find concrete values based on states, so there are no meet or join operations
    - They are looking at specific states and deriving concrete values that are dangerous
    - They are not approximating system functionality
  - Other static analysis used approximation because precision is expensive
    - I am curious how large the tested applications were
    - Authors claim that the code was complicated but my assumption is that there was not a lot of code

Questions

Which University has the ***Hard Times Café*** shown to the left?