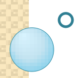
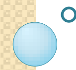


Inside KLEE

Department of Computer Science & Technology
Tsinghua University

Agenda

- About Symbolic Execution and KLEE
- LLVM Assembly Language (Bitcode)
- Memory and Execution States in KLEE
- How KLEE Analyzes Programs



ABOUT SYMBOLIC EXECUTION AND KLEE

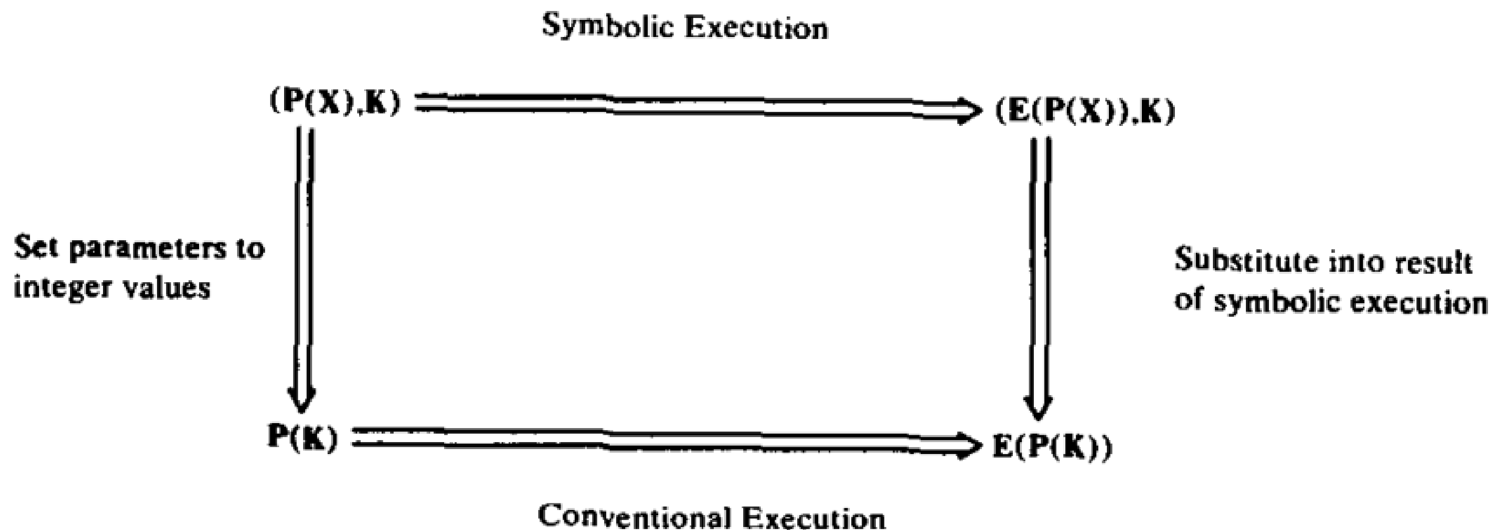
About Symbolic Execution

- Problem
  - Determine what inputs cause each part of program to execute
- Usage
  - Check program reliability
  - Generate test cases automatically
  - Help debugging
- Related techniques
  - Black-box / white-box testing
  - Formal verification
- Key ideas
  - Symbols as input
  - Expressions as variable values
  - Generate a set of <Path Constraints, Output> pairs

About Symbolic Execution

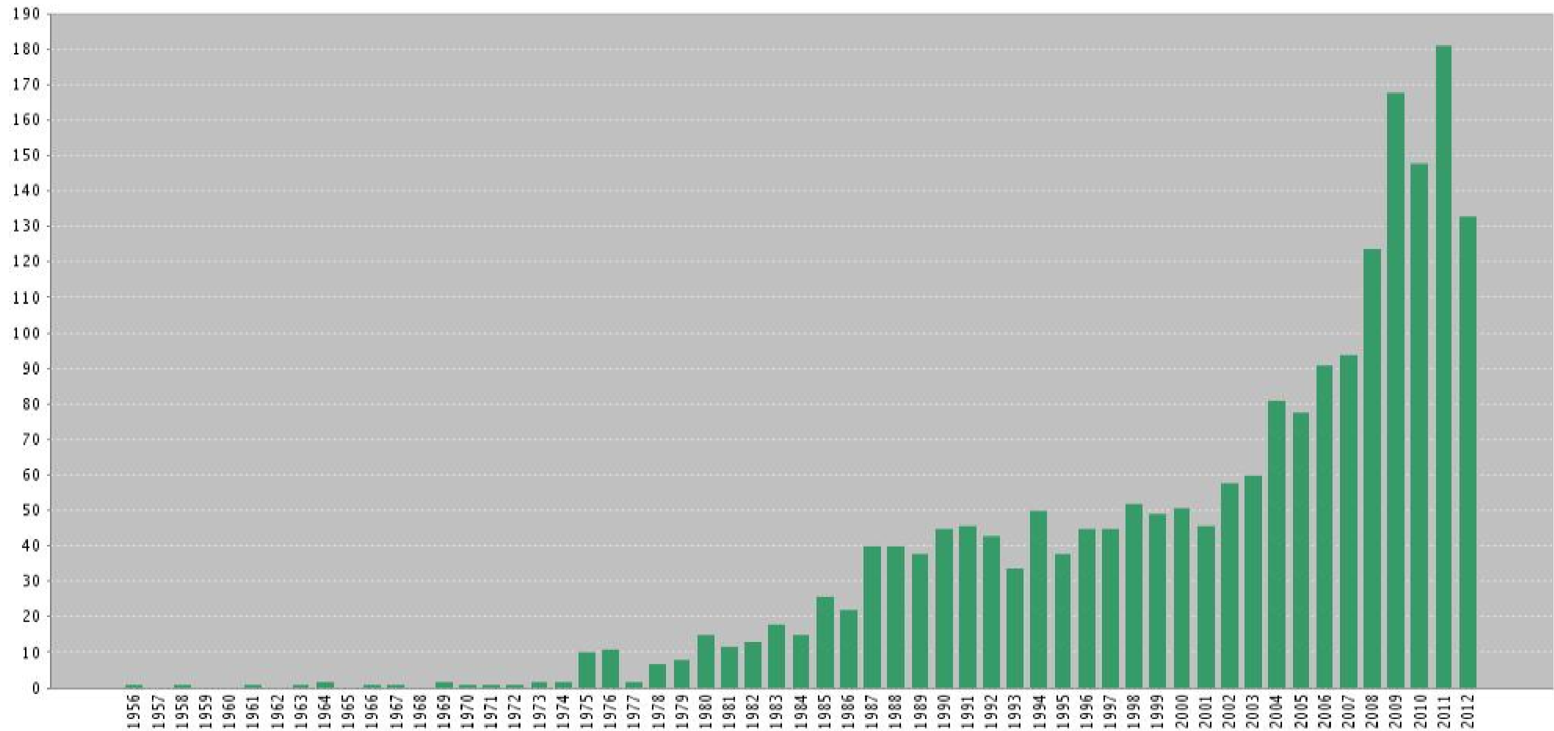
Symbolic Execution vs. Conventional Execution

≈ Algebra vs. Arithmetic



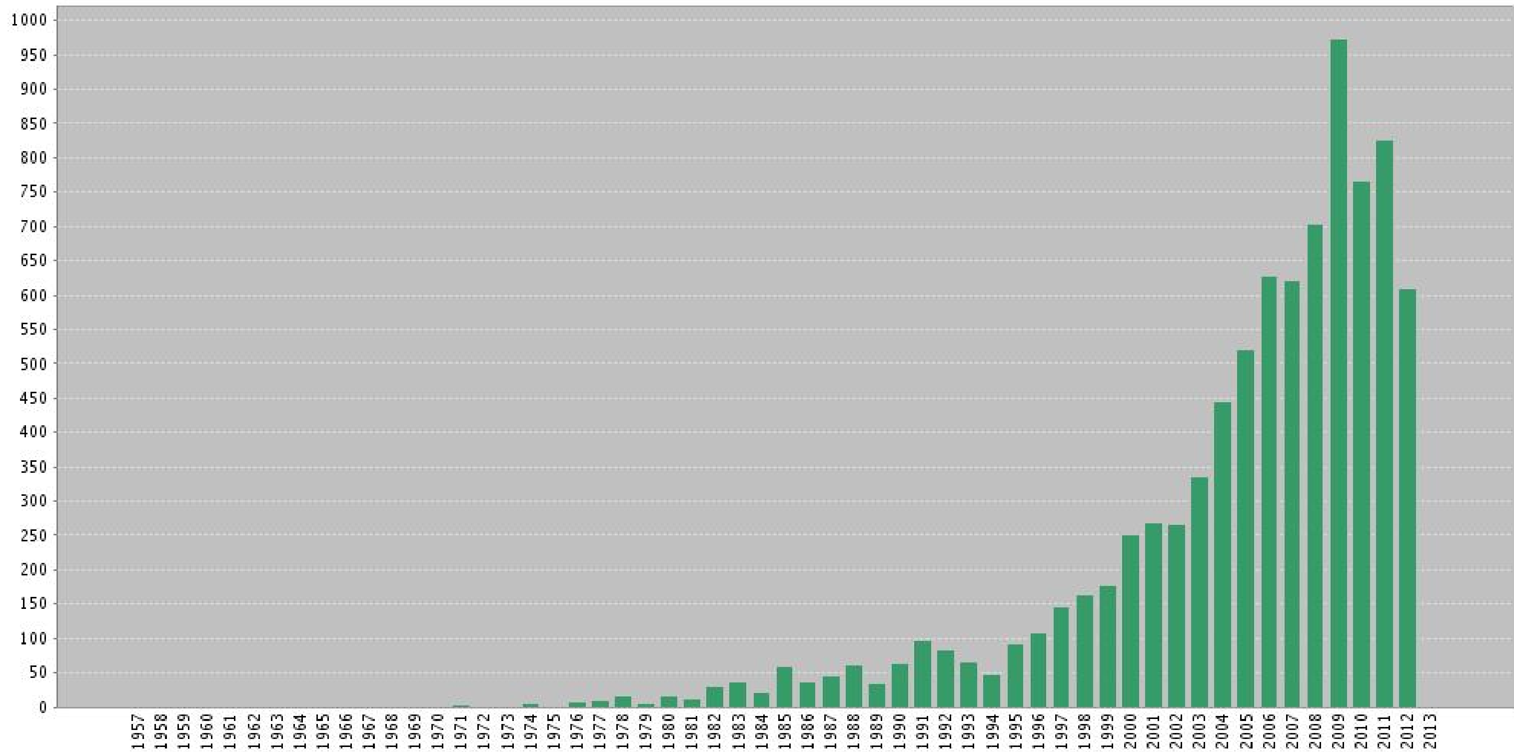
Source: King, J. C. (1976). Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 385–394. doi:10.1145/360248.360252

About Symbolic Execution



Paper Counts（Source：Web of Knowledge）

About Symbolic Execution



Reference Counts （Source： Web of Knowledge）

About Symbolic Execution

- Existing works in academia
  - PathFinder (NASA, TACAS'07)
  - CUTE & jCUTE (UIUC, ISSTA'08)
  - KLEE (Stanford, OSDI'08)
  - CREST (UC Berkeley, AST'10)
  - BitBlaze (UC Berkeley, NDSS'10)
- Applications in industry *
  - Microsoft(Pex, SAGE, YOGI, PREfix)
  - IBM(Apollo)
  - NASA
  - Fujitsu
  - etc.

* Source: Cadar, C., & Godefroid, P. (2011). Symbolic execution for software testing in practice: preliminary assessment. *ICSE'11* (pp. 1–6). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6032591

About KLEE

- OSDI'08
- By Cristian Cadar, Daniel Dunbar, Dawson Engler from Stanford University
- Website : http://KLEE.llvm.org/
- Based on LLVM infrastructure
- Evaluation in the paper
  - Analysis programs in Coreutils (3000~4000 ELOC on average)
  - Time spent in each program is approximately 60min

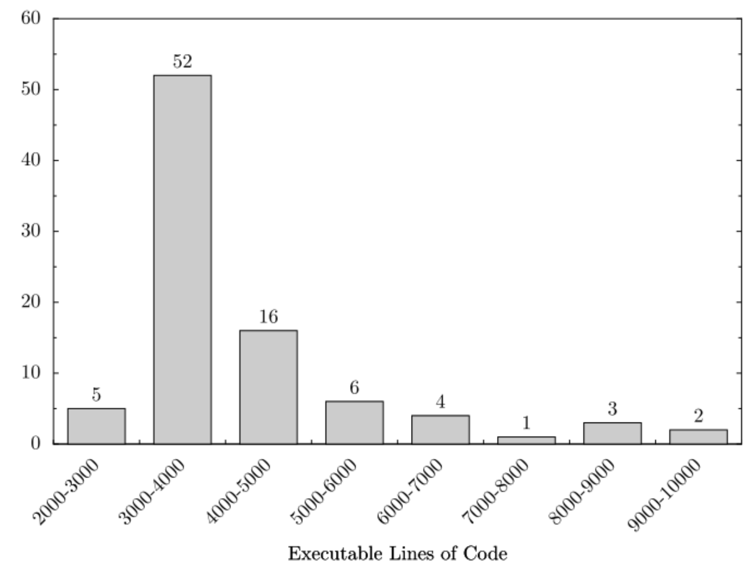


Figure 4: Histogram showing the number of COREUTILS tools that have a given number of executable lines of code (ELOC).

About KLEE

- What is achieved
  - Auto-generation of test cases with high coverage

	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%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

About KLEE

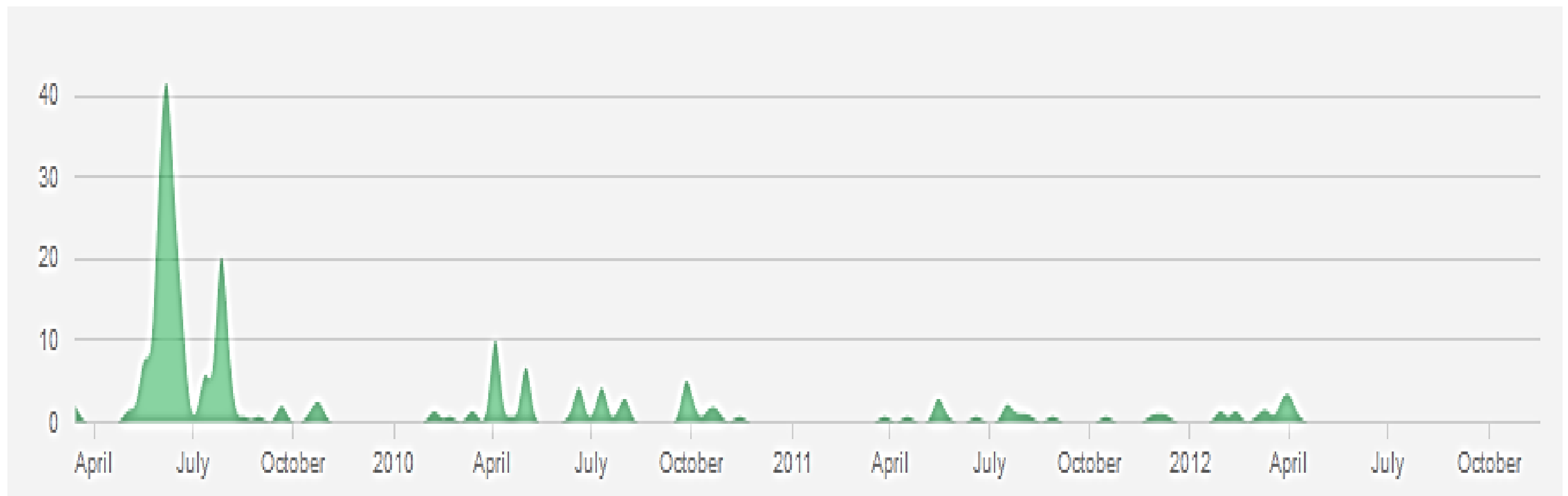
- What is achieved
  - Auto-generation of test cases with high coverage
  - Bug discovery

```
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
```

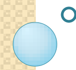
t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

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.

About KLEE



Code Frequency（From Github）



LLVM BITCODE

LLVM Bitcode: An Example

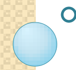
```
int main(int argc, char *argv[])
{
   int i, sum = 0;
   for (i = 0; i < 10; i++)
      sum += i;
   return sum;
}
```

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
%3 = alloca i8**, align 8
%i = alloca i32, align 4
%sum = alloca i32, align 4
store i32 0, i32* %1
store i32 %argc, i32* %2, align 4
store i8** %argv, i8*** %3, align 8
store i32 0, i32* %sum, align 4
store i32 0, i32* %i, align 4
; <label>:4                ; preds = %11, %0
%5 = load i32* %i, align 4
%6 = icmp slt i32 %5, 10
br i1 %6, label %7, label %14
```

On-stack storage (alloca)

Local Variables (%2)

type (i32*)



MEMORY AND EXECUTION STATES IN KLEE

Expressions in KLEE

- All dynamic data (values of global/local variables and in memory cells) are represented by expressions in KLEE
- Examples
  - Constant expression : 5
  - Non-constant expression : (ADD w32 5 (READLSB w32 0))

KLEE: Memory Representation

- KLEE Use Memory Object and Object State for managing data in memory cells
  - Memory Object records basic information (e.g. base address and size) of a continuous memory block
  - Object State acts as a fixed-size array of expression each of with is 8-bit long
- An injective (but not bijective because of COW) mapping is maintained from Memory Object to Object State

KLEE: Memory Representation

- Each expression (representing a byte) in Object State is in one of the following three states
  - Concrete
  - Known Symbolic
  - Flushed
- A write request to Object State provides two arguments including offset of the byte and data to be written

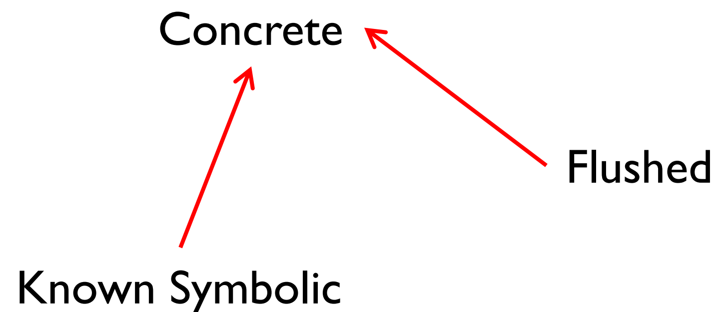
Concrete

Flushed

Known Symbolic

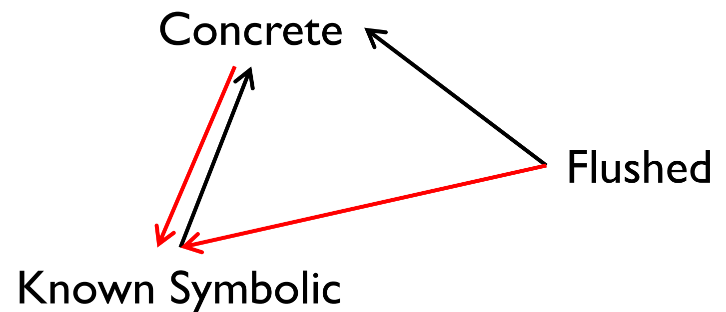
KLEE: Memory Representation

- When offset and data are both constant,
  - the byte written becomes Concrete and
  - read requests afterwards always get the data written this time



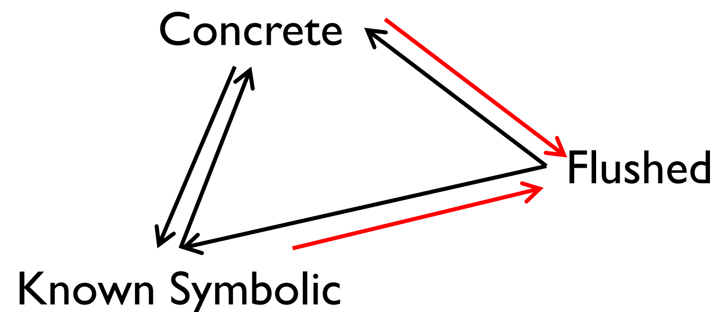
KLEE: Memory Representation

- When offset is constant but data is non-constant,
  - the byte written becomes Known Symbolic and
  - read requests afterwards always get the data written this time



KLEE: Memory Representation

- When offset is non-constant
  - All Concreate and Known Symbolic bytes are flushed to an Update List in the form of write requests with <offset, value> pairs
  - All bytes become Flushed
  - Read from a Flushed byte gets a READ expression
- For read requests with non-constant offset, the bytes are also flushed



KLEE: Execution State

- Execution State represents a snapshot of the program under execution.
- Execution of the program is regarded as transitions among multiple Execution States.

KLEE: Execution State

- What is in an Execution State?
  - Stack

- Stack is a vector of stack frames.
- Each stack frame includes
  - value of local variables, each of which is represented by an expression and
  - Allocation of local storage recorded by a list of Memory Object

KLEE: Execution State

- What is in an Execution State?
  - Stack
  - Memory states

- Memory states are represented by a list of Memory Objects (recording ranges) with their corresponding Object States (recording values).
- Mapping from Memory Objects to Object States is maintained as an Address Space.

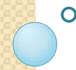
KLEE: Execution State

- What is in an Execution State?
  - Stack
  - Memory states
  - Program counter
- Represented by a wrapper of LLVM Instruction object （Kinstruction）.

KLEE: Execution State

- What is in an Execution State?
  - Stack
  - Memory states
  - Program counter
  - Path constraints

- Path constraints are a set of Boolean expressions recording when this Execution State can be reached



HOW KLEE ANALYZES PROGRAMS

How KLEE works

The main loop:

- Create initial Execution State and add it to the unexplored Execution States list **L**
- while (**L** is not empty) do
- pick up an Execution State **S** from **L**
- execution one instruction of **S**
- if there is any new Execution State generated
- then add generated Execution State(s) to **L**
- else solve path constraints and generate a test case
- done

How KLEE Interprets LLVM Bitcode

- Alloca
  - Format:

    <result> = alloca <type>[, <ty> <NumElements>]

  - When <NumElements> is constant, KLEE will
    - allocate an unused memory range,
    - create an Object State with the same size,
    - initialize the bytes in the Object State with 0xAB,
    - assign the base address of the allocated range to <result> and
    - insert the allocated range into local storage list in the current stack frame (for it will be automatically freed when the function returns)

How KLEE Interprets LLVM Bitcode

- Br
  - Format of branch without condition

    br label <dest>
  - KLEE will
    - set program counter to the first instruction of the target basic block

How KLEE Interprets LLVM Bitcode

- Br
  - Format of branching with condition

    br i1 <cond>, label <iftrue>, label <iffalse>
  - KLEE will
    - Evaluate <cond>,
    - if <cond> is a tautology or a contradiction
      - then branch without condition and add the Boolean expression (Eq <cond> T/F) to path constraints
      - else create a copy of the Execution State, branch to different basic block in different Execution State and add corresponding Boolean expression to the path constraints

How KLEE Interprets LLVM Bitcode

- How to copy Execution State
  - State： copy completely
  - Memory state： copy Address Space and share Object States till they are written
  - Path constraints： copy the container and share expressions

How KLEE Interprets LLVM Bitcode

- Object State: How to copy on write (COW)
  - Each Address Space has a cowKey initialized to 1
  - Each Object State has an ownerKey;
  - When Address Space is copied (including cowKey), cowKey of the new Address Space is incremented
  - Each time handling write requests to Object State **S**, KLEE checks cowKey of current Address Space **A** and ownerKey of the **S**
    - If **A**.cowKey = **S**.ownerKey, the request is performed to **S**
    - If **A**.cowKey != **S**.ownerKey, a copy of **S** (say **S'**) is created, **S'**.ownerKey is set to **A**.cowKey and the request is performed to **S'**

KLEE: Optimizations

- Expression simplification
  - Constant expressions are calculated
  - Usage of operators is restricted
    - e.g. do not use Ult、Uge、Slt、Sge in Boolean expressions
- Constraint rewrite

KLEE: Other Issues Addressed

- Model execution environment
  - command line options
  - environment variables
  - standard libraries
- Execution State picking up policy in the main loop

KLEE: Restrictions

- Because of path explosion, symbolic execution engines can hardly traverse all Execution State
- KLEE provides some options to restrict the execution space explored
  - -max-depth
  - -max-fork
  - ……