KLEE

IV101
Samuel Pastva

What is KLEE?

KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure

Why use it?

- Symbolic execution
- Test synthesis
- High code coverage
- Error detection

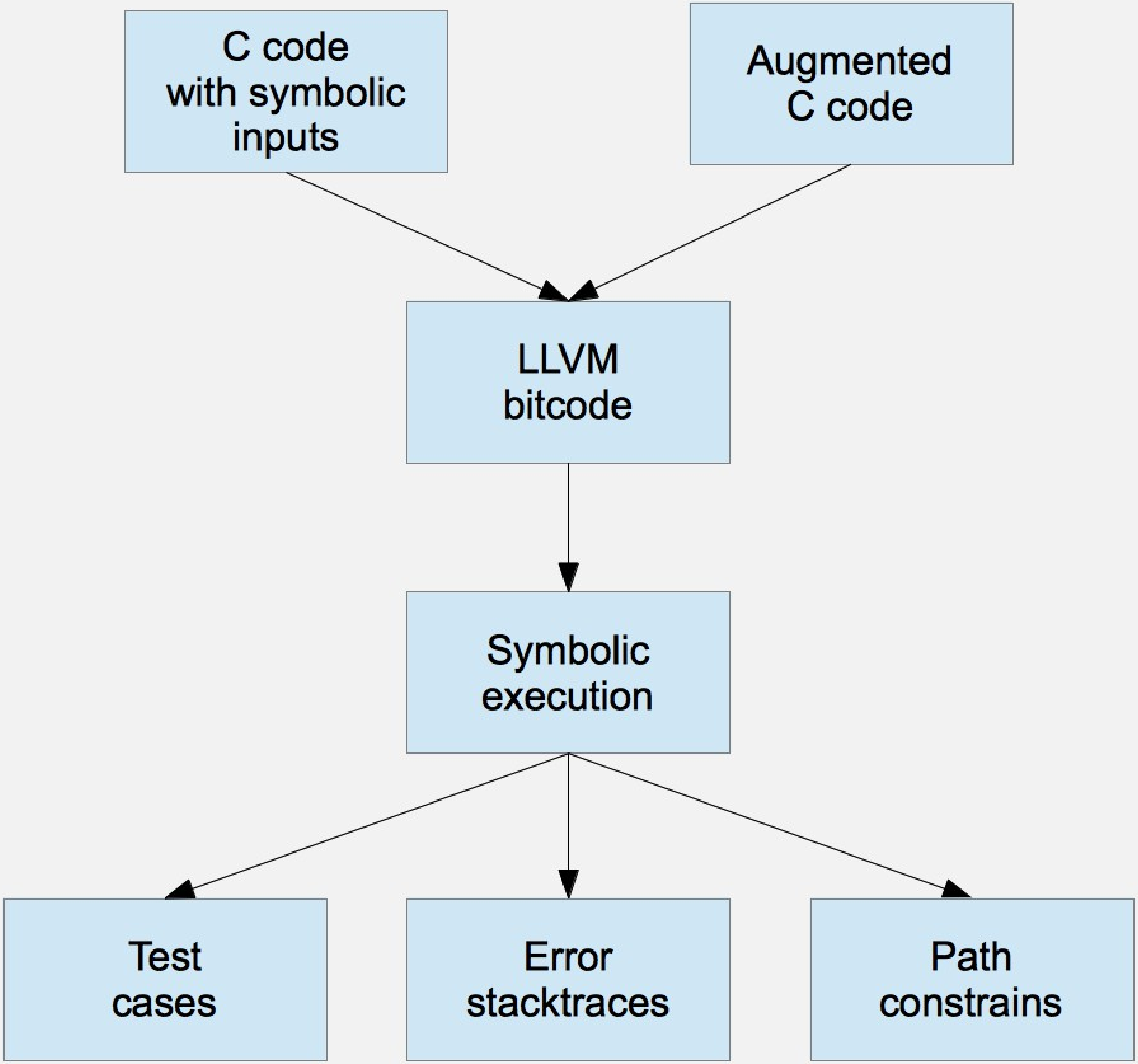
Why not?

No support for:

- Symbolic floating point
- Threads
- Variable size objects

Installation

- Requires LLVM (2.9), STP solver and uclibc (for symbolic POSIX environment)
- Detailed instructions: http://klee.github.io/getting-started/



C code
with symbolic
inputs

Augmented
C code

LLVM
bitcode

Symbolic
execution

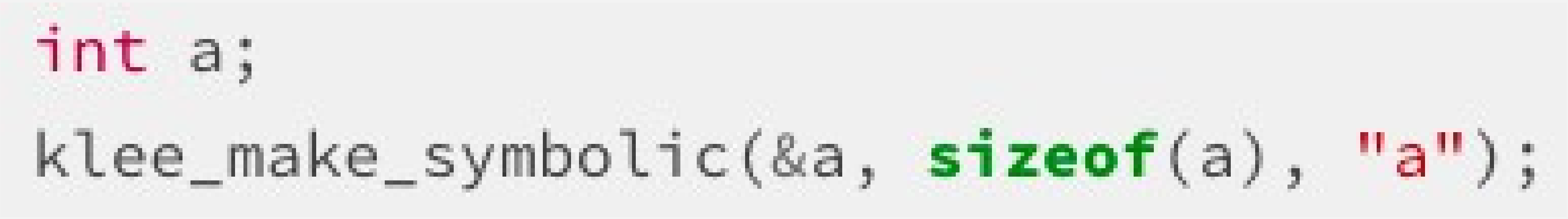
Test
cases

Error
stacktraces

Path
constrains

Augmenting code

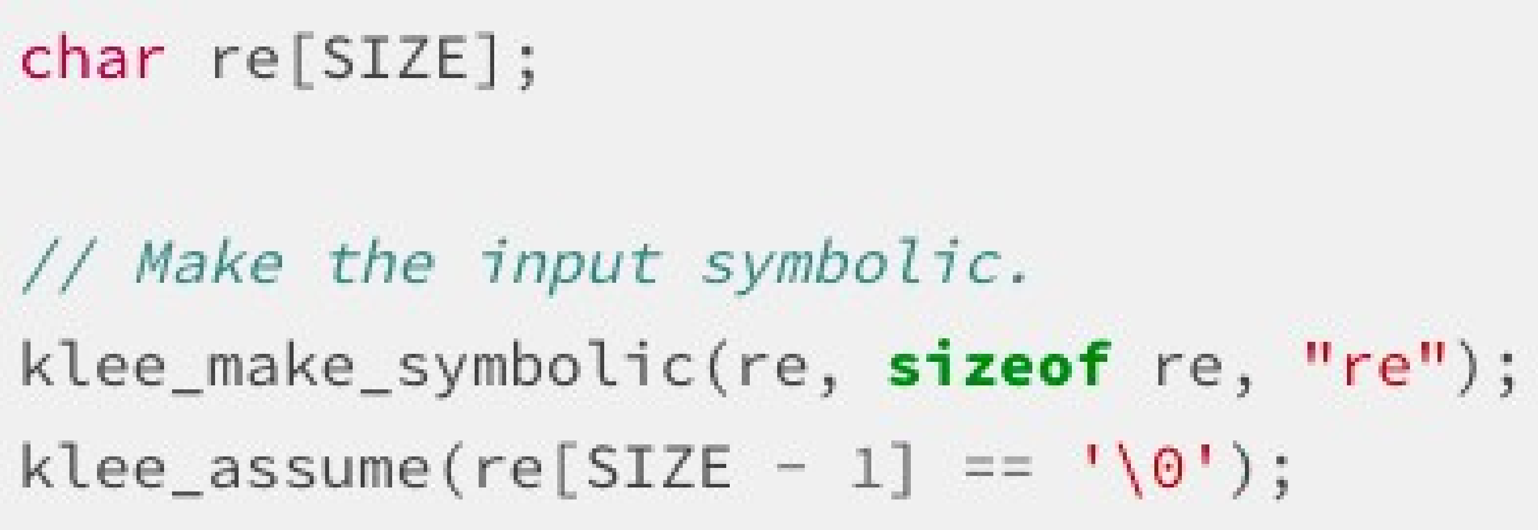
- Call to klee_make_symbolic marks given variable as symbolic
- Variable must have a fixed size



```
int a;
klee_make_symbolic(&a, sizeof(a), "a");
```

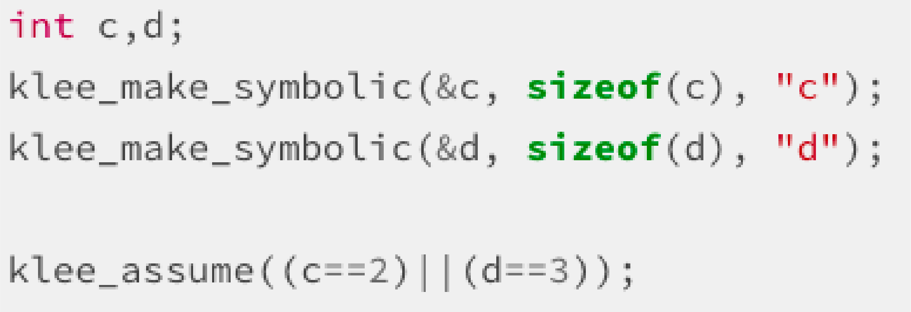
Augmenting code

- Call to klee_assume adds additional constrains to current path
- Reflect conditions not enforced in code
- Cut off paths that are not interesting (speedup)

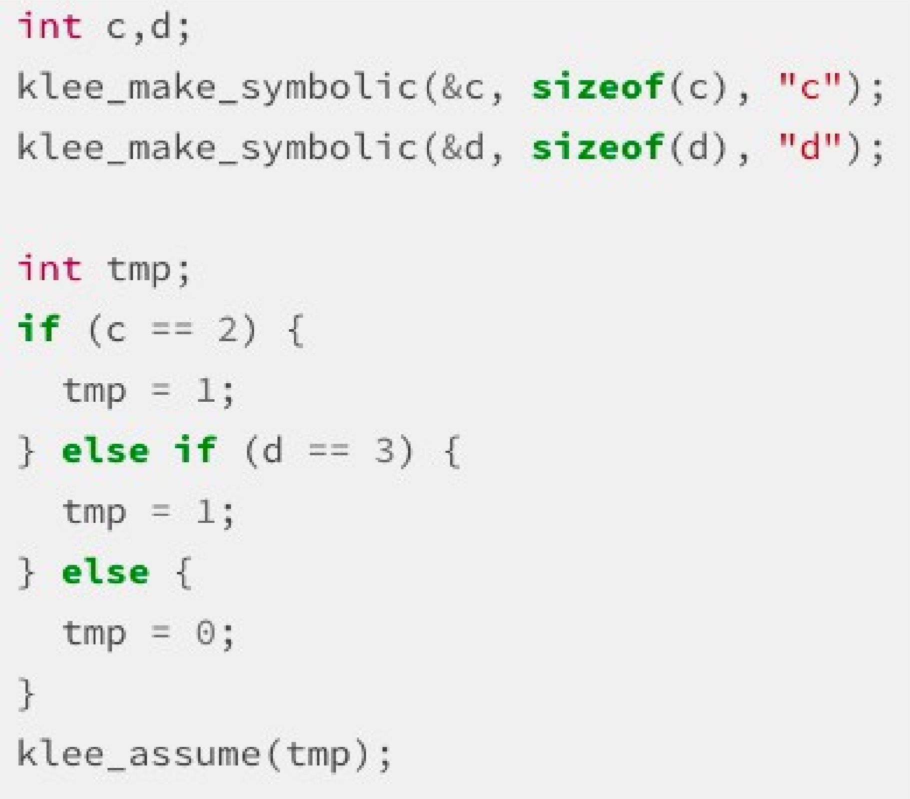


```
char re[SIZE];

// Make the input symbolic.
klee_make_symbolic(re, sizeof re, "re");
klee_assume(re[SIZE - 1] == '\0');
```

Caveats of klee_assume



```
int c,d;
klee_make_symbolic(&c, sizeof(c), "c");
klee_make_symbolic(&d, sizeof(d), "d");

klee_assume((c==2)||(d==3));
```



```
int c,d;
klee_make_symbolic(&c, sizeof(c), "c");
klee_make_symbolic(&d, sizeof(d), "d");

int tmp;
if (c == 2) {
  tmp = 1;
} else if (d == 3) {
  tmp = 1;
} else {
  tmp = 0;
}
klee_assume(tmp);
```

Symbolic input

- Does not require modification of code
- Requires KLEE POSIX Runtime
- Symbolic command line arguments

```
--sym-args size size ...
```

- Symbolic file input and standard input

```
--sym-files count size
```

- Symbolic standard output

```
--sym-stdout
```

- Optionally, environmental failures (full disk, etc.) can be also simulated for most of I/O operations

KLEE – internal structure

- Bitcode is executed on custom virtual machine
- Every execution maintains it's path condition
- On branch, constrain solver (STP) is used to derive new path conditions for every possible outcome and execution is forked
- On every dangerous operation (pointer dereference, division...), constrain solver is used to check if any invalid value is possible

KLEE – limiting execution

- Symbolic execution does not have to terminate (infinite loop, state space is too large)
- KLEE execution can be constrained by memory or by time

  --max-memory=size, --max-time=minutes
- In such cases, the code coverage can be increased by using various search heuristics

Search heuristics

- DFS – Usually lower memory consumption but also lower code coverage
- Random Path – Favors executions with fewer previous forks (avoids starvation and infinite loops)
- Non-uniform Random – Uses random search with custom distribution based on some execution property (instruction count, depth, query cost...)
- Random
- Multiple search heuristics can be interleaved in round robin fashion to achieve more robust results

Testing

- Each execution of KLEE creates new directory that contains test file for every explored path and every error
- Test execution can be easily automated by compiling the binary with libkleeRuntest
  - Each execution of such binary requires a test case specified by KTEST_FILE env. variable
- Alternatively, each test case can be translated to human readable form by ktest-tool utility

Error analysis

- KLEE tracks several types of errors
  - Invalid memory access
  - Double or invalid free
  - Failed assertions
  - Division by zero
- Each detected error is represented by a test case, stacktrace and additional information (heap structure)

Testing Coreutils/Busybox with KLEE

- Average code coverage over 90% with (usually) just
  - 2 short command line arguments
  - 1 long command line argument
  - 2 input files
- 18% greater than any previous test suite
- three 15 years old bugs found
- Checking for inconsistencies with Busybox revealed a lot of incorrect/undefined behaviour

Further work

- KleeNet – verification of sensor networks
- Cloud9 – Distributed symbolic execution engine based on KLEE
- GKLEE – Concolic verification and Test generation for GPUs
- Several experimental extensions of KLEE provide partial support for symbolic floating point values, C++ or parallelism with deterministic interleaving

Demo Time