

1. Software testing

input generation / environment, configuration setting
code coverage (stmt, branch, path)
stmt cv is used in klee (>90%, 80% manual case)
recent testing stories

- Javascript story (<http://ejohn.org/blog/javascript-testing-does-not-scale/>)
- Northeastern blackout

2. symbolic execution

Q: Why is it better than dynamic bug detection?

Why is it better than static analysis?

An example

```
void main(int argc, char** argv){
  x=atoi(argv[1]);
  If(x==16){
    Assert(false);
  }
}
```

EXE:

Symbolic(x);

&x → x3x2x1x0

Fork

Add x==16 to the constraint table

Add x!=16 into another constraint table

Assert check

Advantage:

1. efficient (some times);
2. no false positive; no pointer alias issue.
3. more coverage than dynamic tools

Step:

1. instrumentation
2. symbolic representation and propagation (how to handle pointer)
3. fork
4. constraint solver
5. bug checking or normal exit
6. re-execute with concrete generated input

Special issue:

How to handle environment; how to handle pointer.

this paper

1. better symb execution (experiences from EXE)
2. handle environment better
3. results: coreutils, 80,000 LOC

two targets of klee:

1. statement coverage
2. every dangerous operation (dereference, assertion)

example:

```
--max-time 2 --sym-args 1 10 10
--sym-files 2 2000 --max-fail 1 tr.bc
```

1. instrumentation

EXE: source code: ...

KLEE: LLVM byte code

2. symbolic representation (how to handle pointer)

Symbol table, indexed by concrete address

EXE: term and formula (ripple-carry adder)

Everything is bitvector, boolean, array of bitvector

Use tree (BDD tree) (shrink space by ...)

KLEE: ...

Propagate through execution $v=e$

2.5 for pointer always use $A[xx]$

EXE: move A array into symbol table

What about $A[i]$ with I a symbol? What about symbolic pointer p?

No easy way to handle this, $a[bsym]$ (a need to figure out)

(KLEE's solution ...)

What if A is also symbol? (no way ...)

Have 'object' sense

3. branches and fork

Adding constraints to child processes according to branch conditions

EXE: use OS fork

KLEE: per object

3.5 How to decide which 'processes' to run

(which state/paths to explore first?)

A: start from root and traverse the tree. Randomly decide go to T/F paths.

Essentially, this favors those states high in the tree and also avoid starvation.

4. constraint solver

Optimization:

Independent xx

Caching results

Implied value concretization

5. bug checking or normal exit

6. re-execute with concrete generated input

7. environment

* system call (?)

modeled

if the argument is concrete, do it concrete

if the argument is not concrete

one level, N symbolic files, each has specified length of symb characters

* libc function

this is a design choice: do we write model for libc function or system call

where we put this interface decides

- what implementation errors will we check

(at some point, we need to stop checking things ...)

* how to re-execute with the generated inputs on a real system?

challenge: how to make a system call fail (using ptrace)

detailed experiments:

Q: Why the performance could be better than random testing some times?

A: the part before branch does not need to be run twice

* gcov measures stmt coverage

* cross checking: put mod, mod_opt, write a main and assert(res1==res2)

future:

concurrent programs

Q: why not concurrent programs?

Security

Concrete-symbolic testing