Program Testing
Symbolic Execution

Yue Duan

Outline

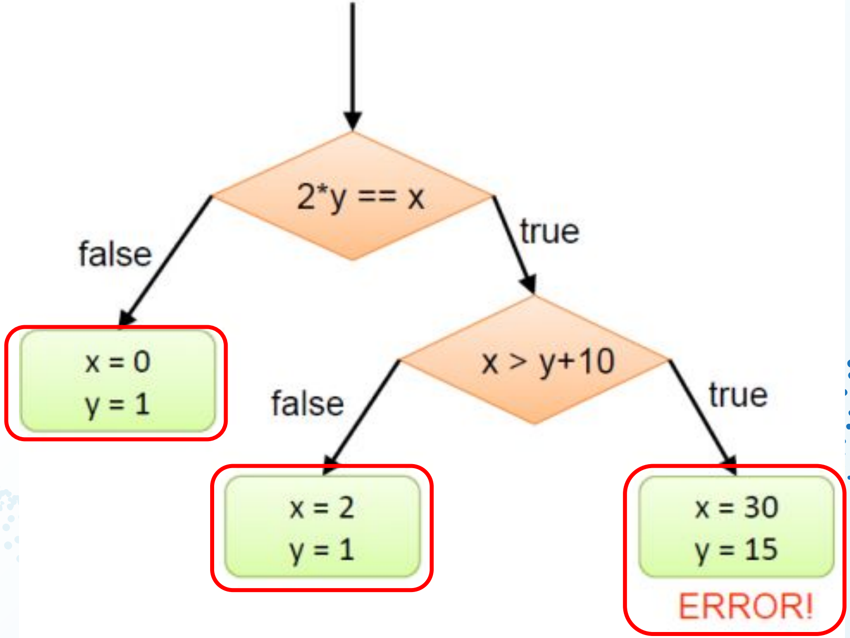
- Introduction to symbolic execution
- Research paper:
    - KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs
    - Automated Whitebox Fuzz Testing

Introduction to Symbolic Execution

- Symbolic Execution
  - assumes **symbolic values** for inputs rather than obtaining **actual inputs** as normal execution of the program would
  - arrives **expressions** in terms of those symbols for expressions and variables in the program
  - collects **constraints** in terms of those symbols for the possible outcomes of each conditional branch
  - aims to **explore all paths** in the program by generating specific inputs

Introduction to Symbolic Execution

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



Problems

- Problem 1: Path explosion
  - states grow exponentially
  - unbounded loop iterations

```
void testme_inf () {
        int sum = 0;
        int N = sym_input();
        while (N > 0) {
            sum = sum + N;
            N = sym_input();
        }
}
```

Problems

- Problem 2: Unsolvable formulas
  - non-linear computation is general hard to solve
  - solver could take too long

```
int twice (int v) {
        return (v*v) % 50;
}
```

Problems

- Problem 3: Environment interactions
  - External function calls and system calls are hard to model
  - Fo efficiency, symbolic execution system usually model
    - file system related calls
    - string operations

```c
int main()
{
  FILE *fp = fopen("doc.txt");
  ...
  if (condition) {
    fputs("some data", fp);
  } else {
    fputs("some other data", fp);
  }
  ...
  data = fgets(..., fp);
}
```

Concolic Testing

- Symbolic + concrete execution
  - run symbolic execution dynamically
  - execute the program on some concrete input values
- Example
  - generate random input: x=22, y=7
  - execute the program both concretely and symbolically

```
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

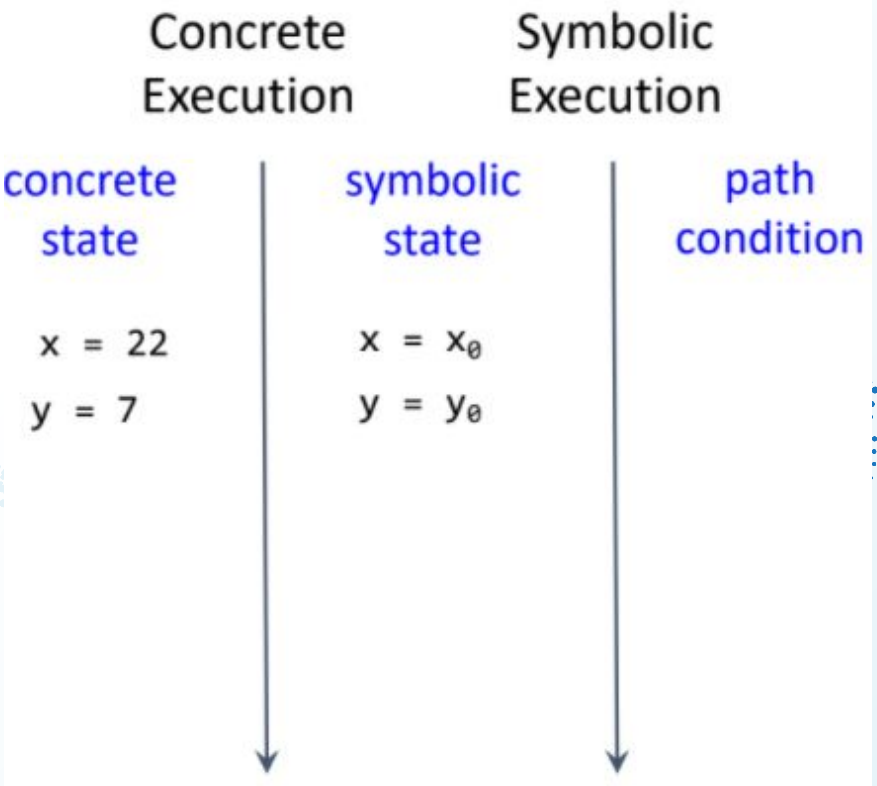
Concolic Testing

- Example (Cont.)
  - concrete execution takes the 'else' branch on Ln.7
  - symbolic execution generates the path constraint: x != 2*y
  - a negation will make the path constraint: x == 2*y
  - solve the path constraint and get a new test input: x=2, y=1
  - test the program with the new input

```
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concolic Testing

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



Concolic Testing

```
int foo(int v) {
    return 2*v;
}

void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
x = 22	$x = x_0$	
y = 7	$y = y_0$	
z = 14	$z = 2*y_0$	

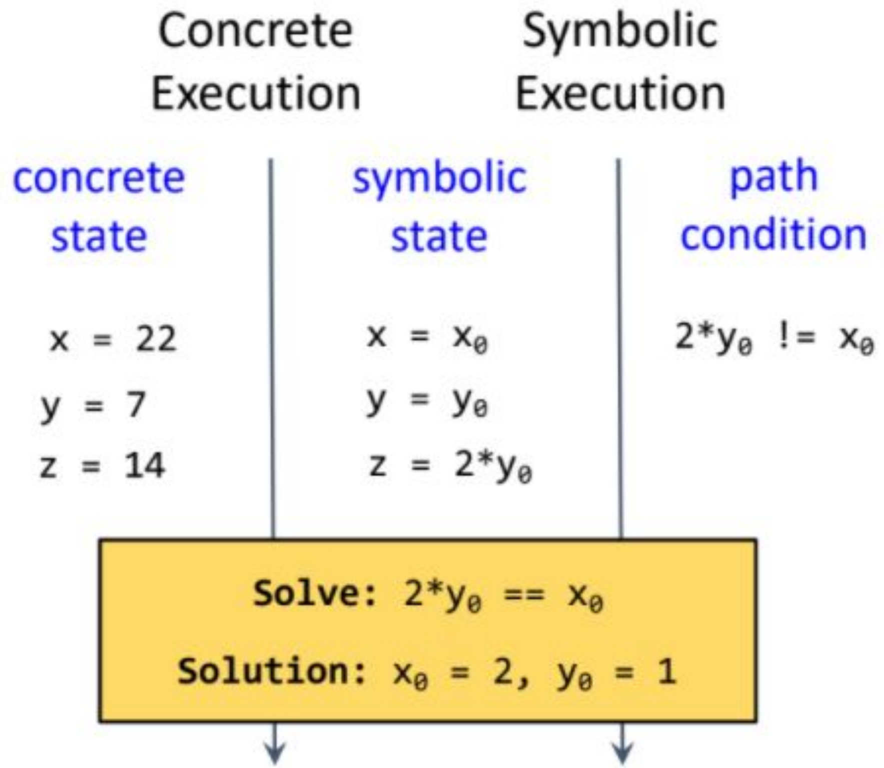
Concolic Testing

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

	Concrete Execution		Symbolic Execution
	concrete state	symbolic state	path condition
	x = 22	$x = x_0$	$2*y_0 \; != \; x_0$
	y = 7	$y = y_0$	
	z = 14	$z = 2*y_0$	

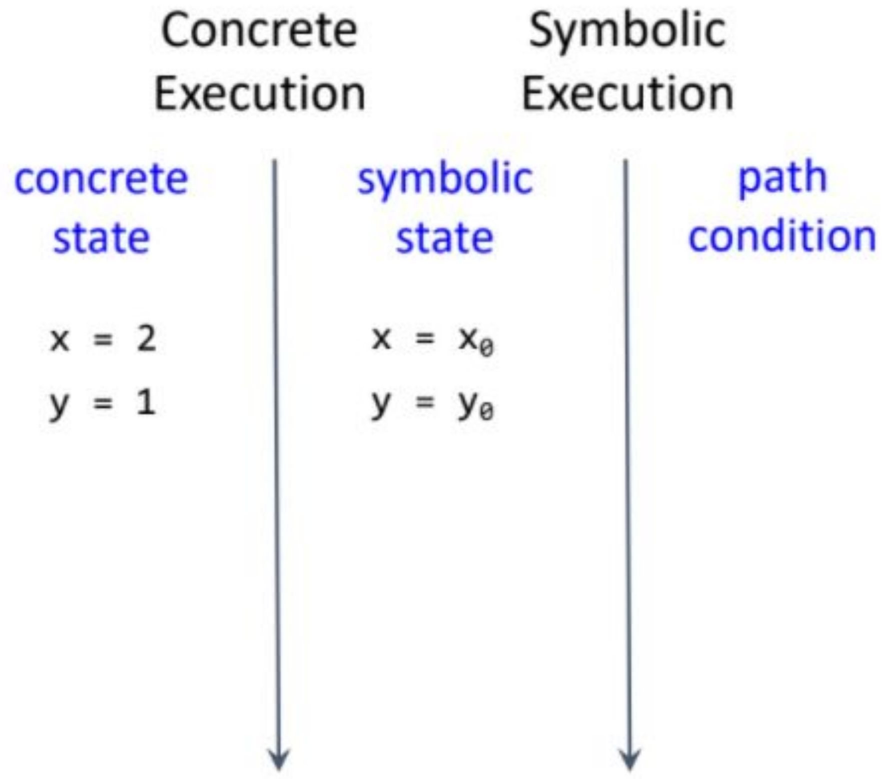
Concolic Testing

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



Concolic Testing

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



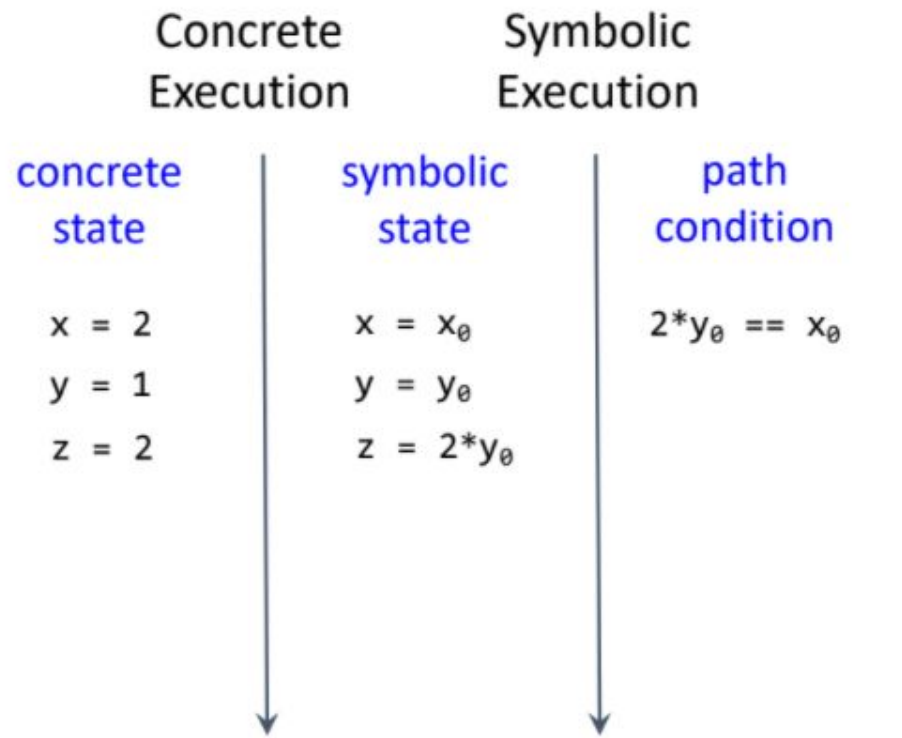
Concolic Testing

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```

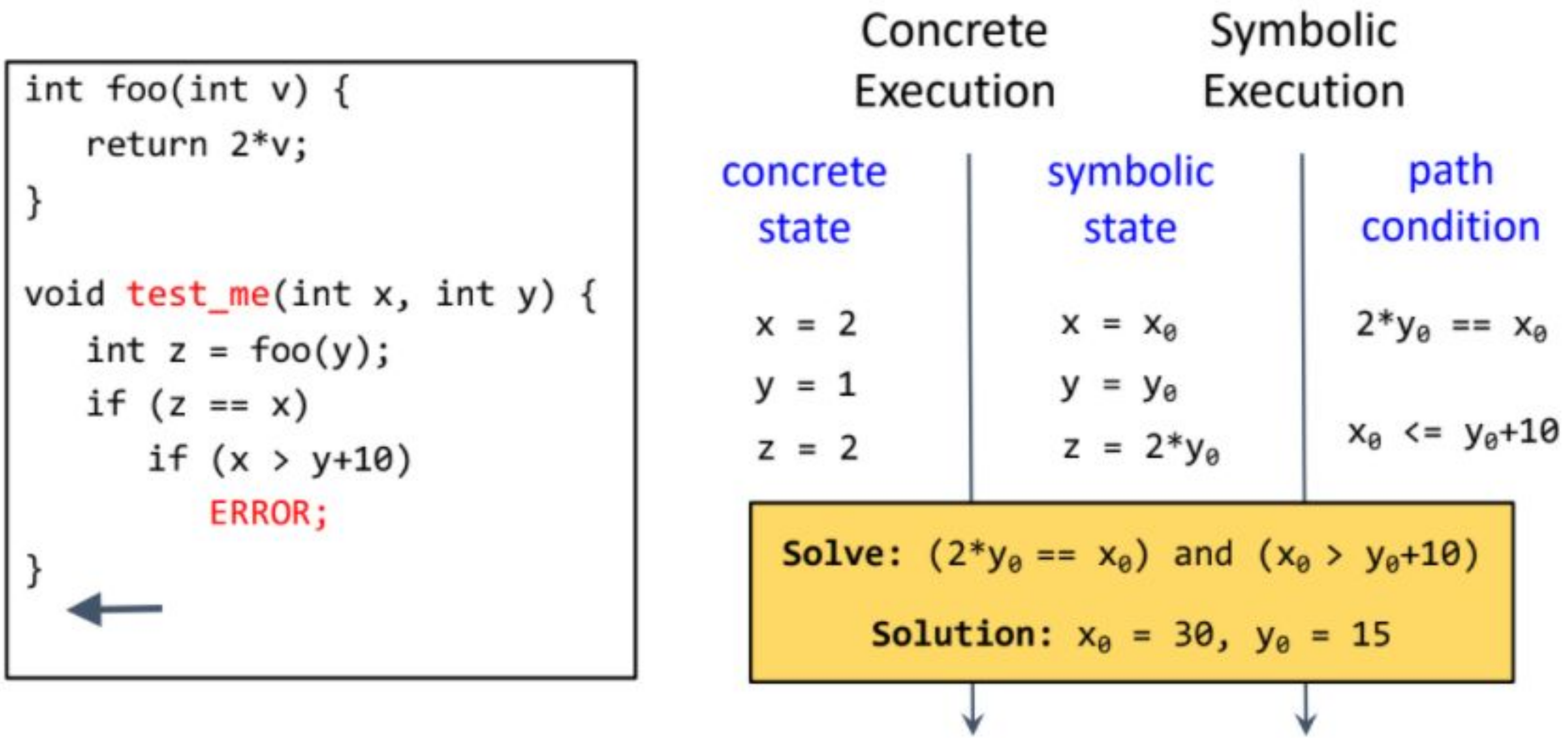
Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 2	$x = x_0$	
y = 1	$y = y_0$	
z = 2	$z = 2*y_0$	

Concolic Testing

```
int foo(int v) {
   return 2*v;
}

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
         ERROR;
}
```



Concolic Testing



Concolic Testing

- Benefits:
  - solve complex formulas
    - x == (y*y) mod 50, unsolvable if x and y are both symbolic
    - if some value is concretized, then it becomes solvable
  - External library call and system call
    - e.g., fd = open(filename)
    - set filename to its concrete value “/tmp/abc.txt”
    - execute the system call concretely
    - set fd to be concrete after the system call return

Online V.S Offline Approaches

- Online
    - encounter a new symbolic branch
    - solve path constraints for both ‘true’ and ‘false’
    - if both feasible, fork the execution states
- Offline
    - trace-based approach
    - choose an input, execute the program and collect execution trace
    - compute path contracts from the trace
    - negate each conjunct, solve the new path constraint and generate new inputs
    - start again

Online V.S Offline Approaches

- Pros and Cons

	Online	Offline
Efficiency	High	Low
Implementation difficulty	High	Low
Symbolic State	Quickly exploded	No state management

Implementations

- Dynamic Instrumentation
  - source code needed:
    - compile C/C++ into LLVM bytecode
    - add instrumentation during compilation
  - binary:
    - run in QEMU with two machines (concrete and symbolic)
    - convert TCG IR to LLVM bytecode
- Trace-based
  - collect execution trace using tools such as Pintrace and tracecap
  - convert trace into IR
  - perform analysis on IR

Implementations

- Pure Interpretation or Simulation
  - interpret binary execution and add symbolic execution logic
  - Pros:
    - full control
    - easy to implement
  - Cons:
    - low efficiency (all instructions must be interpreted)

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

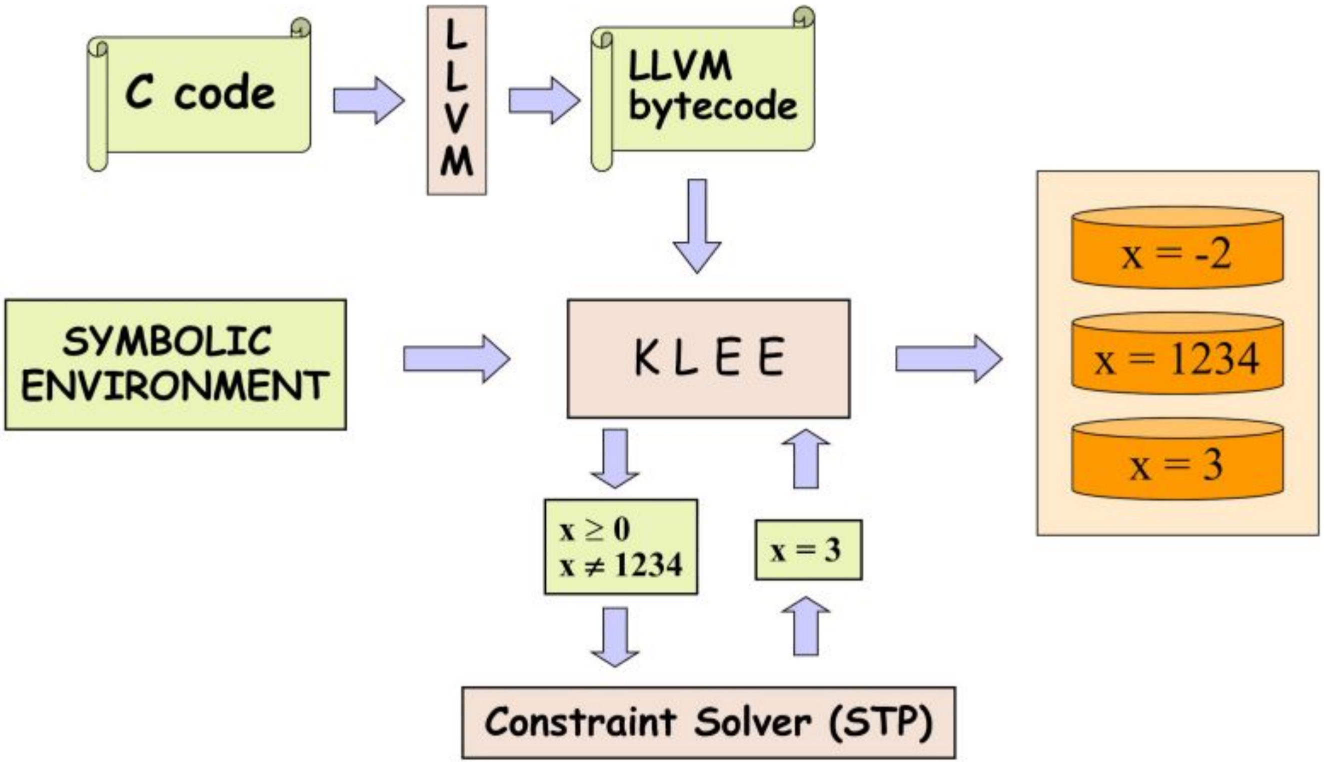
Cristian Cadar, Daniel Dunbar, Dawson Engler

OSDI 2008

Introduction

- Operates on LLVM bytecode
- A **symbolic process** (or **state**) is the state of a symbolically executing process
  - register file, stack, heap, program counter, path condition
  - storage locations(stack, heap, registers) contain **symbolic expressions**
- when symbolic execution counters a branch
  - state is cloned
  - update instruction pointer and path condition accordingly

KLEE Architecture



State Exploration

- Problem:
  - The number of states grow exponentially
- Solution
  - use compact state representation:
    - copy-on-write at the object level
    - heap as an immutable map can be shared among states
    - heap can be cloned in constant time

Path Selection

- Multiple concurrent states, representing different program executions
- Aim:
  - good code coverage
- Problem:
  - which state to run at each step?
- Solution
  - two strategies
    - random path selection
    - coverage-optimized search

Environment Modeling

- Problem:
  - interactions with the environment are complex
- Solution:
  - model semantics and redirects library calls to these models
- Example: symbolic file system
  - single directory with N symbolic files
  - co-exist with real file system
    - when called with concrete file name, will open real file
      - int fd = open(“/etc/fstab”, O_RDNLY);
    - when called with symbolic file name, will form and match each of the N symbolic files
      - int fd = open(argv[1], O_RDNLY);

Evaluation

- Metrics: Line coverage
- Dataset:
  - Coreutils, Busybox, HiStar
- Results:
  - in much shorter time, better coverage than tests manually developed for over 15 years

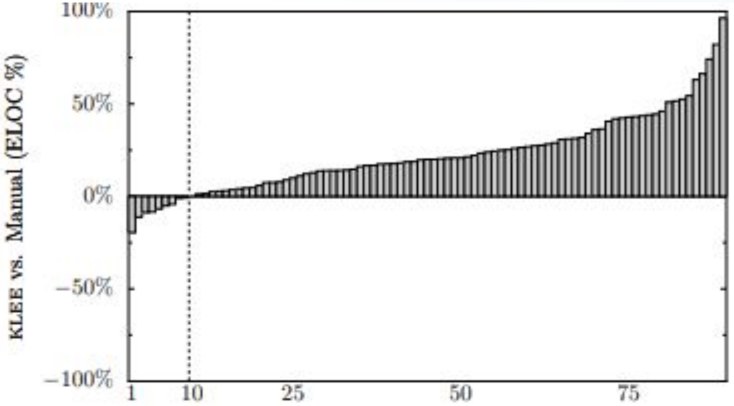


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.

Evaluation

- Found 10 unique bugs in Coreutils
- Found 21 bugs in Busybox
- Found 21 bugs in Minix
- All memory errors

```
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.

Automated Whitebox Fuzz Testing

Patrice Godefroid, Michael Y. Levin, David Molnar

NDSS 2008

Motivation

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) crash();
}
```

input = "good"

I_0 != 'b'

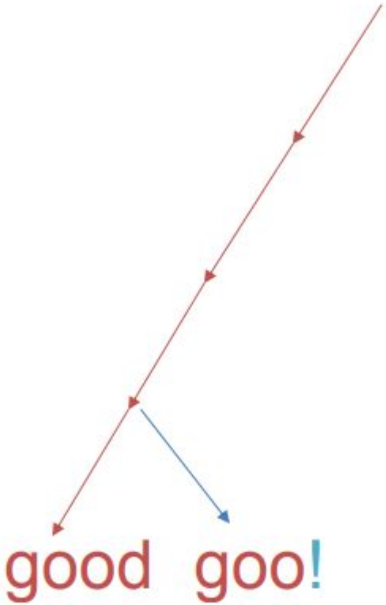
I_1 != 'a'

I_2 != 'd'

I_3 != '!'

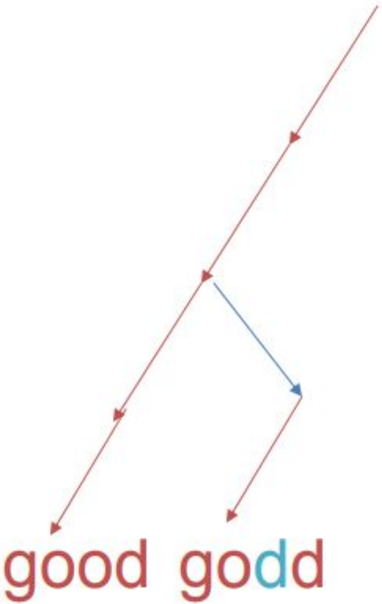
- Traditional trace-based approach
  - Collect constraints from trace
  - create new constraints by negating
  - solve ⇒ new input
  - start again with new input

Motivation



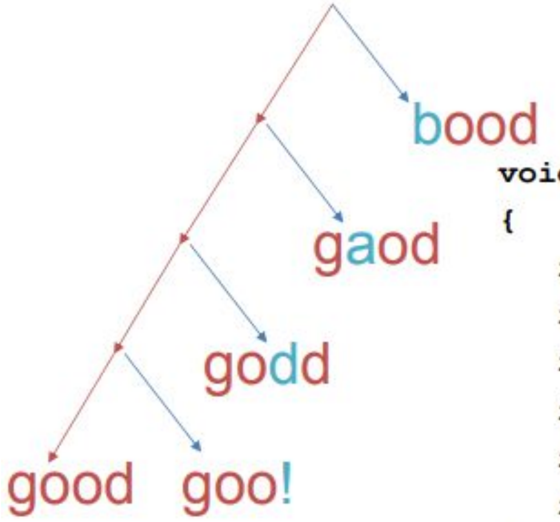

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;   I0 != 'b'
    if (input[1] == 'a') cnt++;   I1 != 'a'
    if (input[2] == 'd') cnt++;   I2 != 'd'
    if (input[3] == '!') cnt++;   I3 == '!'
    if (cnt >= 3) crash();
}
```

Motivation




```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;   I0 != 'b'
    if (input[1] == 'a') cnt++;   I1 != 'a'
    if (input[2] == 'd') cnt++;   I2 == 'd'
    if (input[3] == '!') cnt++;   I3 != '!'
    if (cnt >= 3) crash();
}
```

Key Idea: One Trace Many Tests

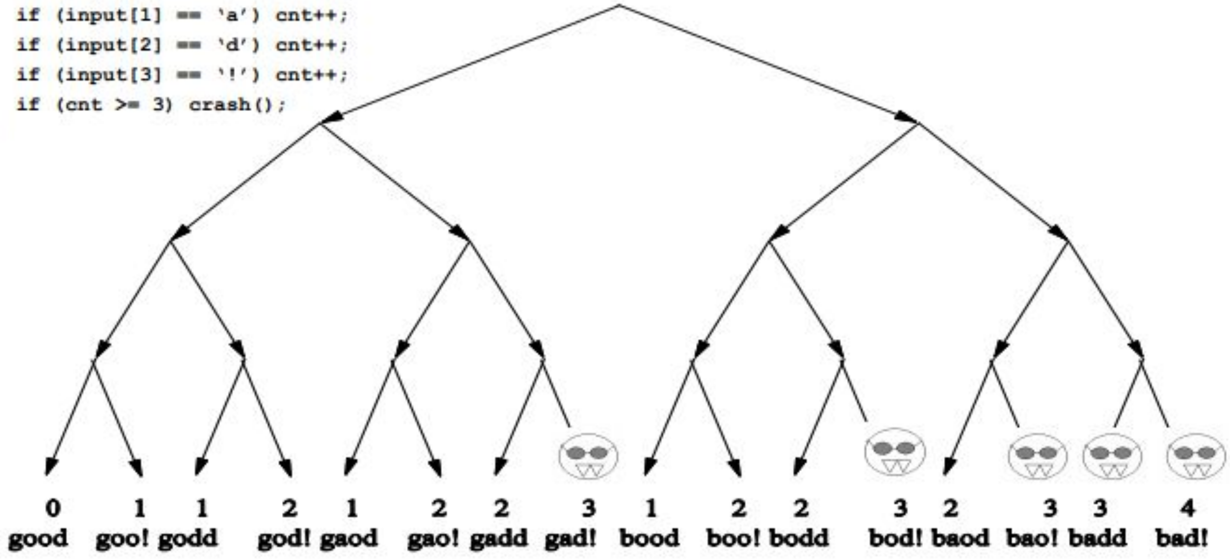



```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;    I0 == 'b'
    if (input[1] == 'a') cnt++;    I1 == 'a'
    if (input[2] == 'd') cnt++;    I2 == 'd'
    if (input[3] == '!') cnt++;    I3 == '!'
    if (cnt >= 3) crash();
}
```

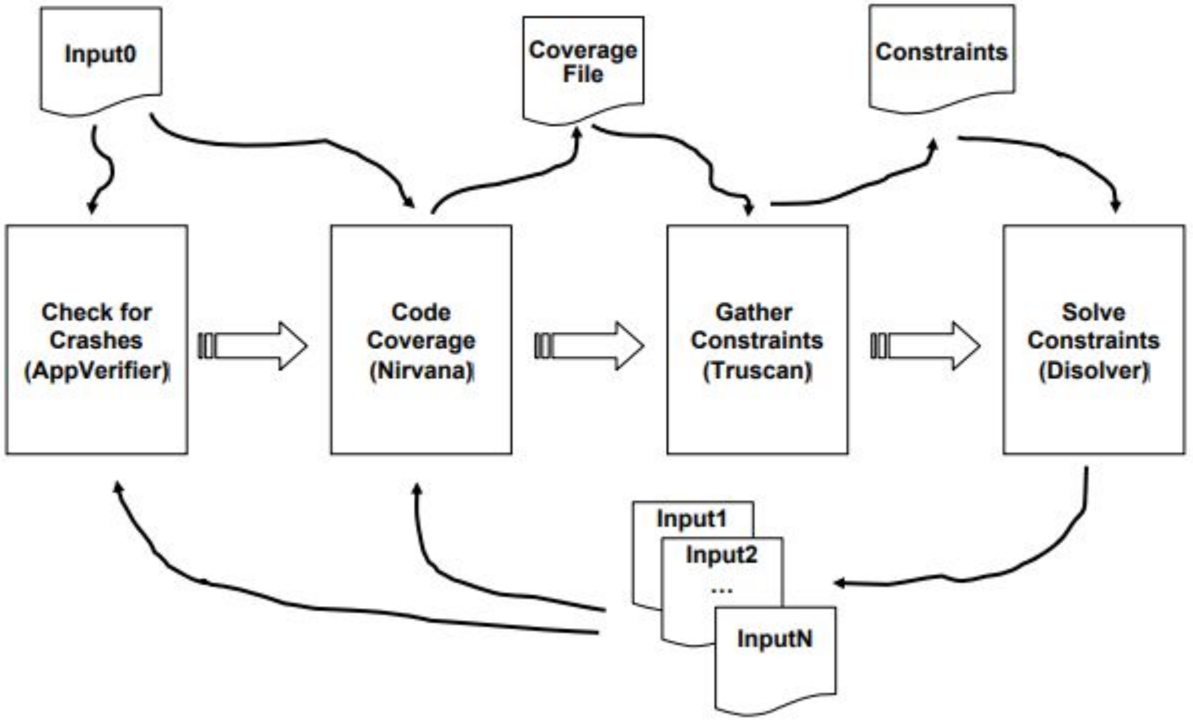
"Generation 1" test cases

Key Idea: One Trace Many Tests

```
void top(char input[4])
{
    int cnt = 0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) crash();
}
```



SAGE Architecture



Case Study

ANI Parsing - MS07-017

Critical, **out-of-band** security patch; affected Vista

Seed file

```
RIFF...ACONLIST
B...INFOINAM....
3D Blue Alternat
e v1.1..IART....
................
1996..anih$...$.
................
................
..rate..........
..........seq ..
................
..LIST....framic
on.........   ..
```

SAGE-generated crashing test case

```
RIFF...ACONB
B...INFOINAM....
3D Blue Alternat
e v1.1..IART....
................
1996..anih$...$.
................
................
..rate..........
..........seq ..
................
..anih....framic
on.........   ..
```

Only **1 in 2^{32} chance** at random!

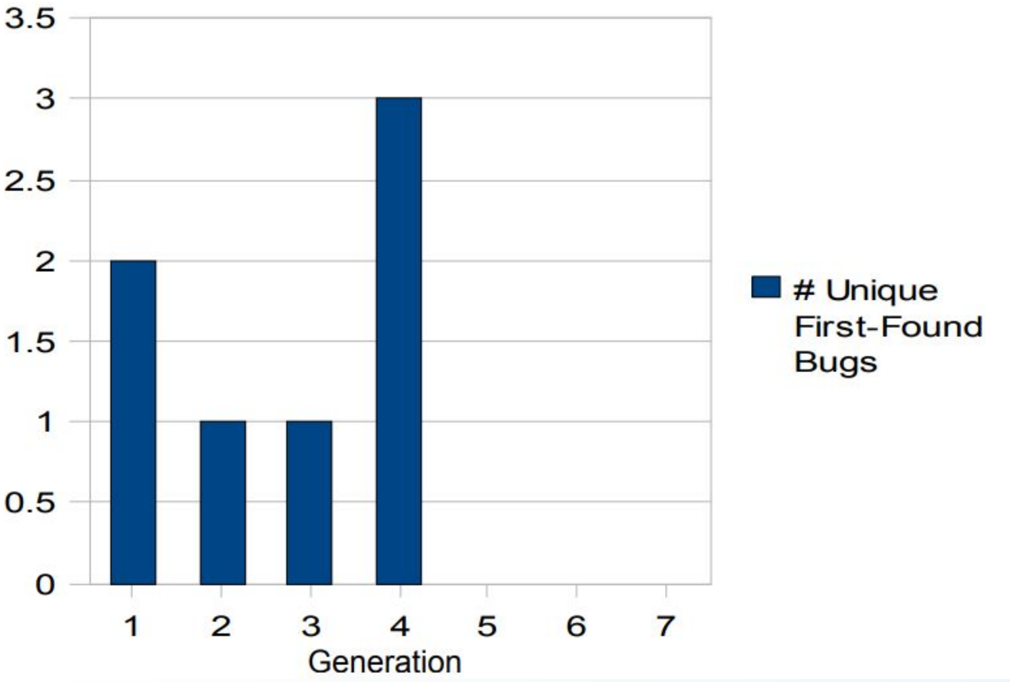
Evaluation

- Very large dataset

App Tested	#Tests	Mean Depth	Mean #Instr.	Mean Size
ANI	11468	178	2,066,087	5,400
Media 1	6890	73	3,409,376	65,536
Media 2	1045	1100	271,432,489	27,335
Media 3	2266	608	54,644,652	30,833
Media 4	909	883	133,685,240	22,209
Compression	1527	65	480,435	634
Office 2007	3008	6502	923,731,248	45,064

Evaluation

- Most bugs found are ‘shallow’



Thank you!

Questions?