symbolic execution

have an emulator/virtual machine

but represent input values as *symbolic variables*

- like in algebra

choose a path through the program, track *constraints*

- what values did input need to have to get here?

then solve constraints based on variables to create real test case

- no solution? impossible path
- find solution? test case

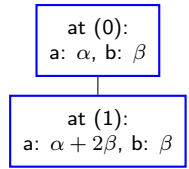
example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```

at (0):
a: α, b: β

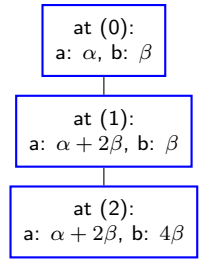
example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



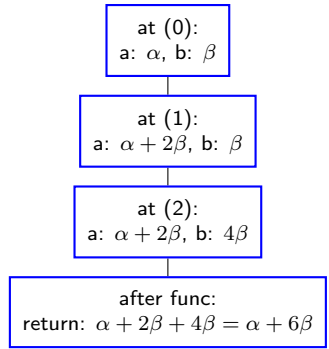
example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



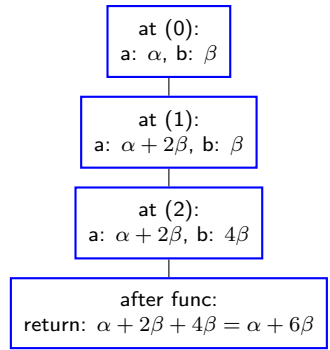
example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



example 0

```
int foo(int a, int b) {
    // (0)
    a += b * 2;
    // (1)
    b *= 4;
    // (2)
    return a + b;
}
```



can express return value of function in terms of arguments

then can solve for possible value of arguments

example: if return $==$ 10, then can enumerate:

$(\alpha, \beta) = (10, 0)$

$(\alpha, \beta) = (4, 1)$

actually doing this

angr is a binary analysis toolkit written in Python

- has Ghidra-like GUI, but not very stable/maintained as far as I can tell

among other things, converts assembly into intermediate form

supports symbolic execution

angr setup

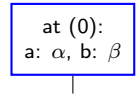
```
import angr
import claripy

p = angr.Project("./example0",
                 load_options='auto_load_libs': False)

foo_addr = p.loader.main_object.get_symbol('foo').rebased_addr
input_a = claripy.BVS('initial_a', 32) # 32-bit bit vector
input_b = claripy.BVS('initial_b', 32) # 32-bit bit vector
init_state = p.factory.call_state(foo_addr, input_a, input_b)
simgr = p.factory.simulation_manager(init_state)
# <SimulationManager with 1 active>
```

angr running

```
print(f"RIP={simgr.active[0].regs.rip} versus {foo_addr:#x}")
    # \textit{RIP=<BV64 0x4011f9> versus 0x4011f9}
print(f"EAX={simgr.active[0].regs.eax}")
    # \textit{RAX=<BV reg_eax_3_32>} (unknown value)
simgr.step()
    # simgr = \textit{<SimulationManager with 1 active>}
simgr.step()
    # simgr = \textit{<SimulationManager with 1 deadended>}
state = simgr.deadended[0]
print(f"EAX={state.regs.eax}")
    # \textit{EAX=initial_a_0_32 + }
    # \textit{     (initial_b_1_32[30:0] .. 0) +}
    # \textit{     (initial_b_1_32[29:0] .. 0)}
state.solver.add(state.regs.eax == 10)
print(state.solver.eval(input_a), state.solver.eval(input_b))
    # \textit{10 0}
state.solver.add(input_b != 0)
print(state.solver.eval(input_a), state.solver.eval(input_b))
    # \textit{4294901754 715838808}
```

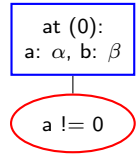
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



example 1

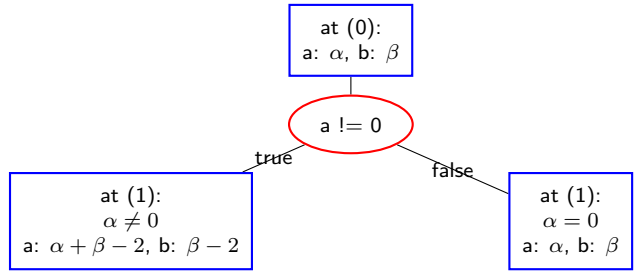
```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



every variable represented as an *equation*

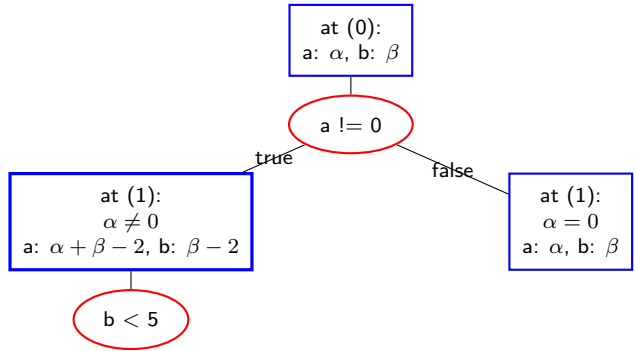
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



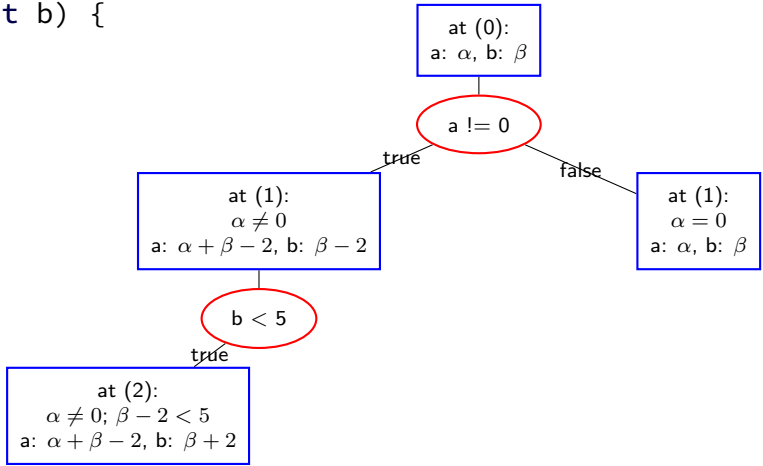
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



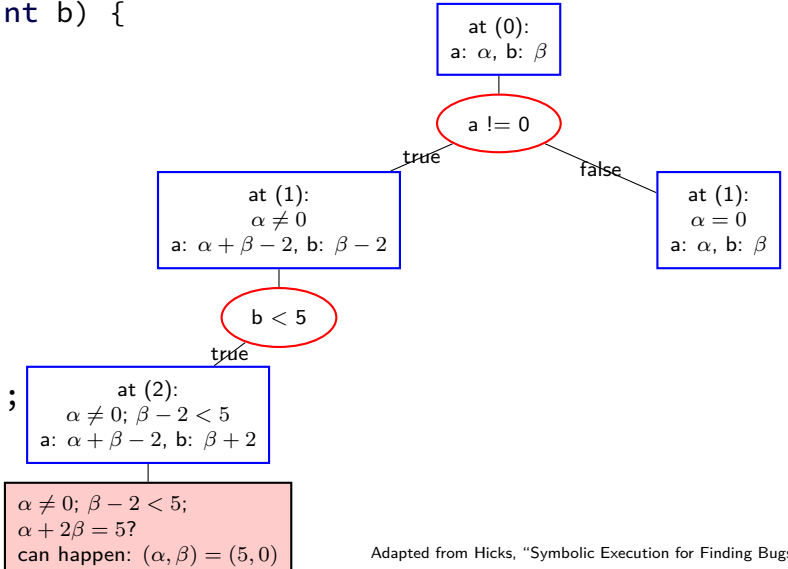
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



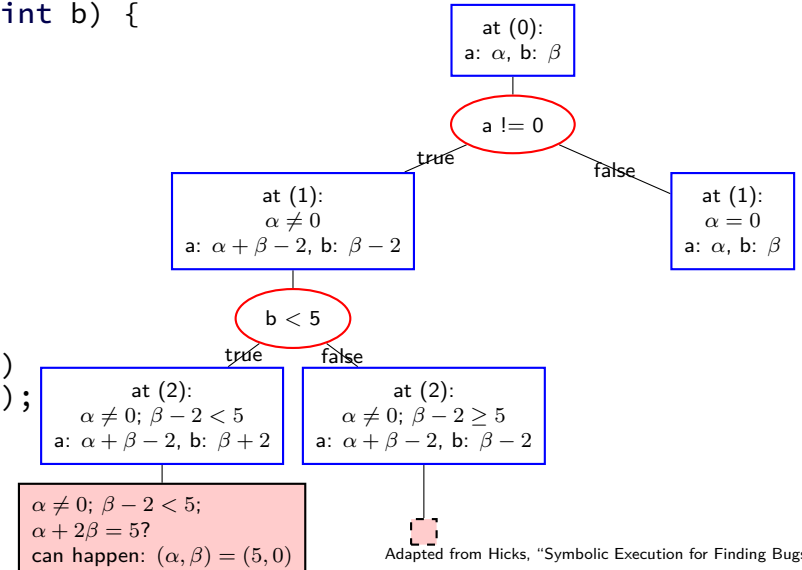
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



example 1

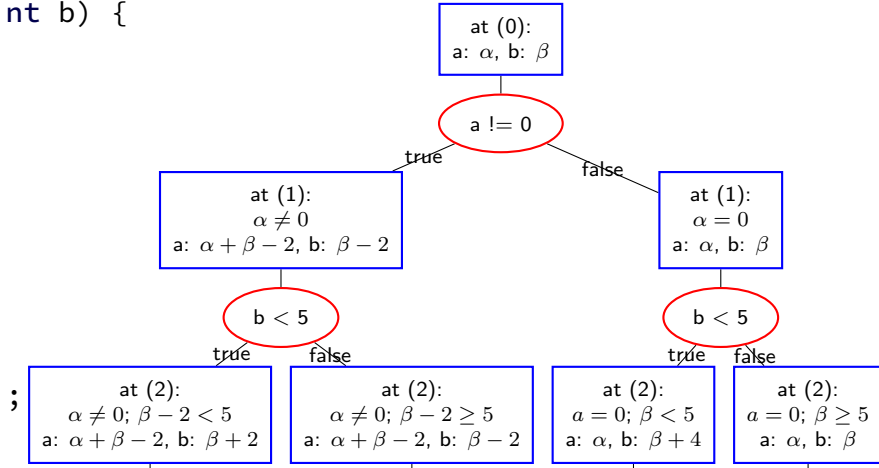
```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  if (a + b == 5)
    INTERESTING();
}
```



example 1 in angr

```
p = angr.Project("./example1", load_options={'auto_load_libs': False})

foo_addr = p.loader.main_object.get_symbol('foo').rebased_addr
INTERESTING_addr = p.loader.main_object.get_symbol('INTERESTING').rebased_
input_a = claripy.BVS('initial_a', 32)
input_b = claripy.BVS('initial_b', 32)
init_state = p.factory.call_state(foo_addr, input_a, input_b)

simgr = p.factory.simulation_manager(init_state)
print("at beginning:", simgr)
simgr.explore(find=INTERESTING_addr)
print("after explore:", simgr)
for state in simgr.found:
    found_a = state.solver.eval(input_a)
    found_b = state.solver.eval(input_b)
    print(f'(a, b) = ({found_a}, {found_b})')
```

```
after explore: <SimulationManager with 4 deadended, 4 found>
(a, b) = (0, 1)
(a, b) = (0, 5)
(a, b) = (1, 2)
```

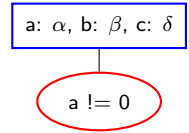
example 2

a: α, b: β, c: δ

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```

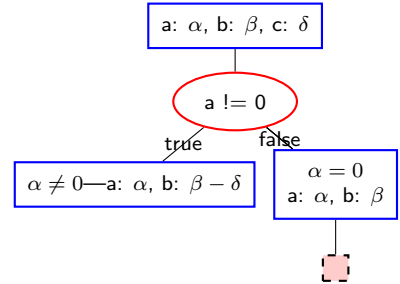
example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



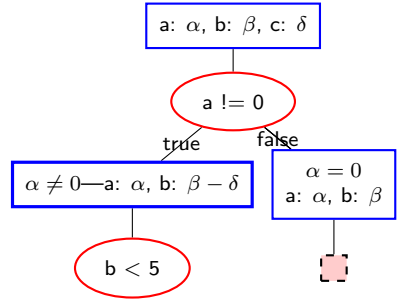
example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



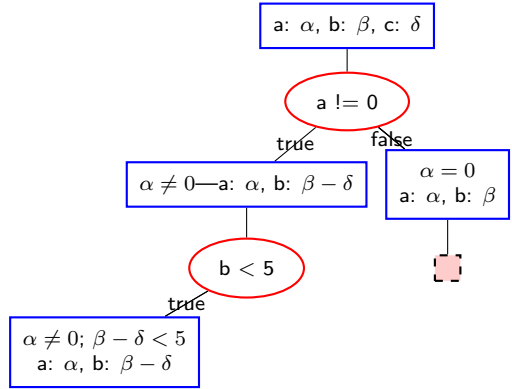
example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



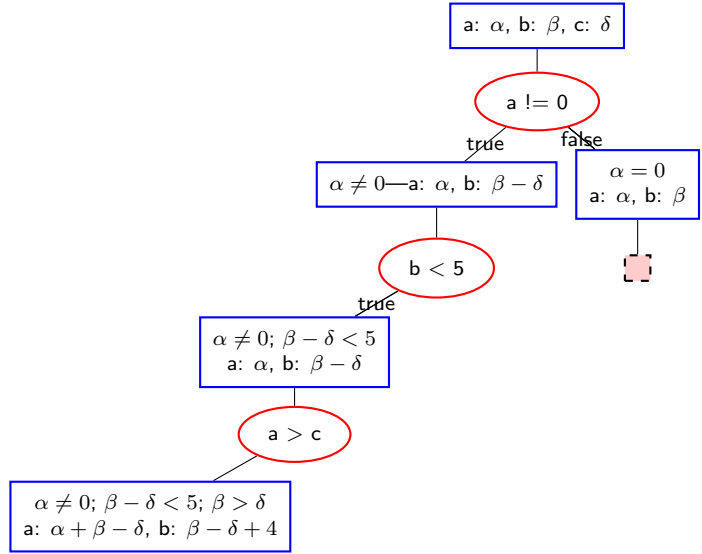
example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



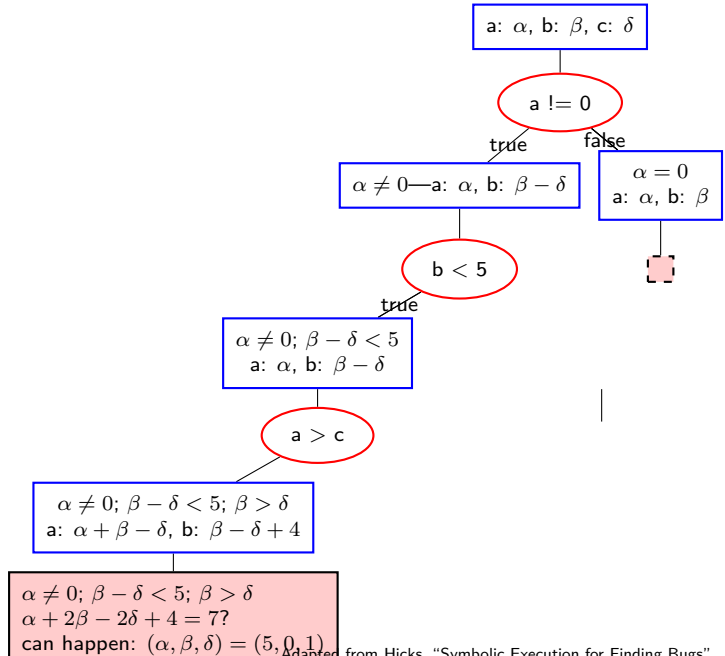
example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



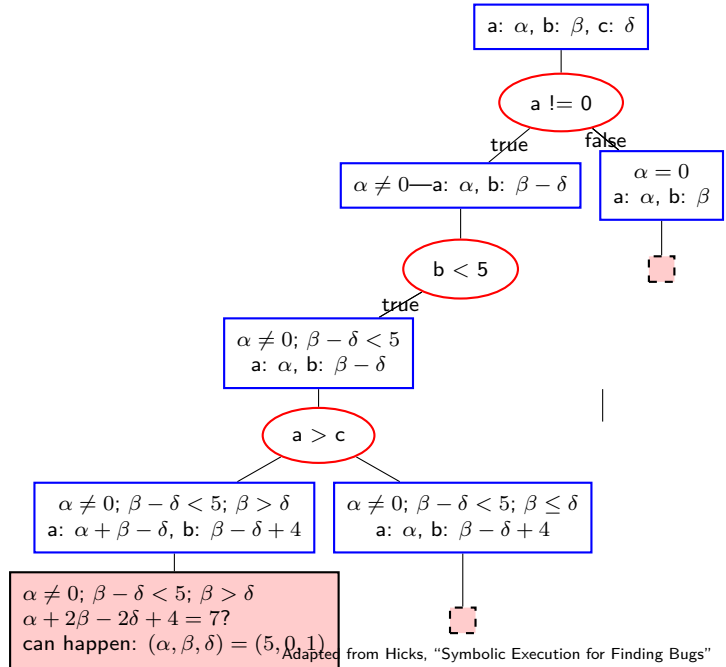
example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



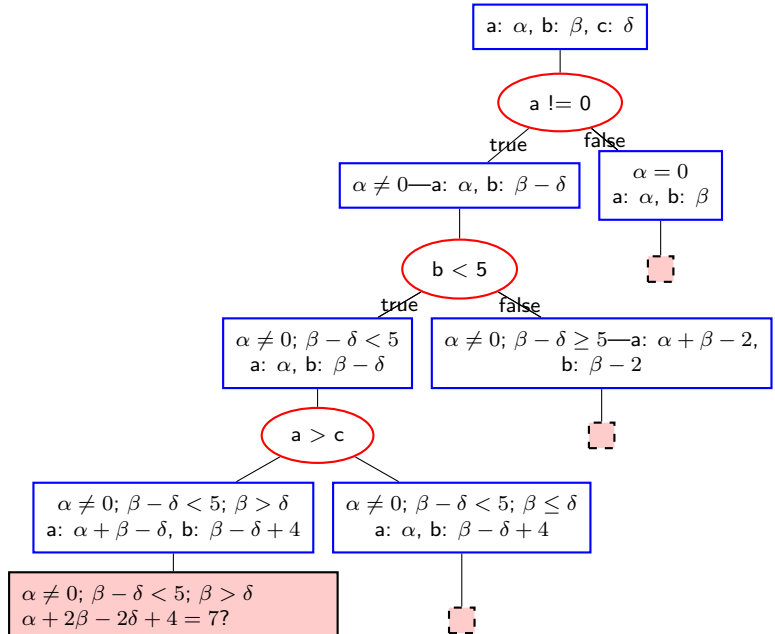
example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



example 2

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
  if (a != 0) {
    b -= c; // W
  }
  if (b < 5) {
    if (b > c) {
      a += b; // X
    }
    b += 4; // Y
  } else {
    a += 1; // Z
  }
  if (a + b != 7)
    INTERESTING();
}
```



executing what...

angr: executing machine code

- works on any existing application
- handles programs with mix of many programming languages
- allows mixing with “fast”(ish) emulation
- can find issues resulting from reused uninitialized values/etc.
- more convenient for reverse engineering/exploits

other systems (exmaple: KLEE): work on ‘intermediate representation’

- simplifies equations dramatically
- idenitfies out-of-bounds/etc. accesses as error typically (even if it never triggers something exploitable now)
- tractable without approximation for much larger programs

aside: concolic

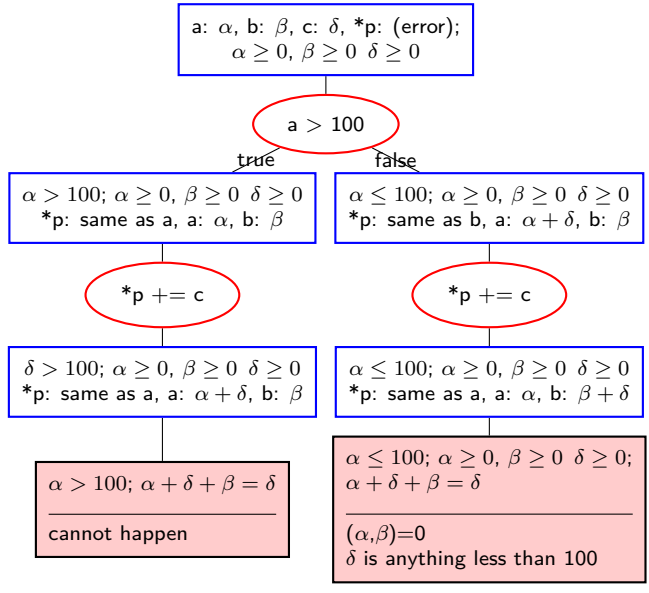
mixing symbolic and ‘concrete’ execution

use symbolic execution, but then produce concrete inputs

systems that combine both sometimes called ‘concolic execution’

example 3

```
unsigned a, b;
void foo(unsigned c) {
    int *p;
    if (a > 100) {
        p = &a;
    } else {
        p = &b;
    }
    *p += c;
    assert(a + b == c);
}
```



exercise

```
void example(unsigned x, unsigned y) {
    if (x > y) return;
    x = x + y;
    assert(x + y + 1 > y);
}
```

1: to see if the assertion is meant, the equation we should solve (if initial values of x, y, are X, Y)?

2: what is an input that fails the assertion? (hint: integer overflow)

equation solving

can generate formula with bounded inputs

can always be solved by trying all possibilities

but actually solving is *NP-hard (i.e. not generally possible)*

luck: there exists solvers that are *often* good enough

...for small programs

...with lots of additional heuristics to make it work

tricky parts in symbolic execution

dealing with pointers?

- one method: one path for each valid value of pointer

solving equations?

- NP-hard (boolean satisfiablity) — not practical in general
- "good enough" for small enough programs/inputs
- ...after lots of tricks

how many paths?

- $< 100\%$ coverage in practice
- small input sizes (limited number of variables)

symbolic memory?

```
int array[10];
void example(int a, int b) {
    assert(a >= 0 && b >= 0 && a < 10 && b < 10);
    int *p = &array[a]; int *q = &array[b];
    *p += 1;
    *q -= 1;
    if (*p == *q) {
        ...
    }
}
```

question: how to deal with *p, *q accesses?

lots of branches?

```
int array[10];
void example(int a, int b) {
    assert(a >= 0 && b >= 0 && a < 10 && b < 10);
    int *p = &array[a]; int *q = &array[b];
    *p += 1;
    *q -= 1;
    if (*p == *q) {
        ...
    }
}
```

separate branch for all 10 possible values of x, y

gets correct answer — but 100x extra branches to explore?

angr ‘concretization’ strategies

angr’s design: choose specific set of real addresses for reads and writes

problem: really common to have thousands/millions of possibilities

default policy:

choose any of range of 1024 possibilities for read
only choose maximum address for write, except for annotated values

higher-level strategies

KLEE (2008; OSDI): symbolic execution for bugfinding

works from LLVM bytecode, not assembly

track *memory objects* that pointers point to

- angr can't really do this — info not available in assembly
- really want to have 'pointers' that aren't integers

if pointer goes outside intended memory object: found a bug!

- deliberate caveat: find overflow bugs, not *exploits*

if pointer can have multiple objects: fork execution for each one

- like it was an if statement

solve equation for all possible indices in memory object

running example

```
int array[2];
int foo(int x, int y){
    array[x] = 0x10;
    array[y] = 0x20;
    return array[x] + array[y];
}
```

exercise: possible return values?

in angr (0)

```
p = angr.Project('mem-demo.so', load_options={'auto_load_libs': False})
x = claripy.BVS('x', 31).zero_extend(1) # 0 sign bit + [31 unknown bits]
y = claripy.BVS('y', 31).zero_extend(1) # 0 sign bit + [31 unknown bits]
foo_func = p.loader.find_symbol('foo').rebased_addr
foo_prototype = angr.sim_type.SimTypeFunction(
    args=(angr.sim_type.SimTypeInt(), angr.sim_type.SimTypeInt()),
    returnty=angr.sim_type.SimTypeInt()
)
st = p.factory.call_state(foo_func, x, y, prototype=foo_prototype)
st.add_constraints(x >= 0, x < 2, y >= 0, y < 2)
sm = p.factory.simulation_manager(st, save_unconstrained=True)
sm.explore()
print(sm)
    # <SimulationManager with 1 deadended>
print(sm.deadended[0].regs.eax)
    # <BV32 0x40>
```

in angr (0): warnings

```
WARNING  | The program is accessing register with an unspecified value.
           This could indicate unwanted behavior.
WARNING  | angr will cope with this by generating an unconstrained symbolic variable
           and continuing. You can resolve this by:
WARNING  | 1) setting a value to the initial state
WARNING  | 2) adding the state option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS},
              to make unknown regions hold null
WARNING  | 3) adding the state option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS},
              to suppress these messages.
WARNING  | Filling register 76 with 4 unconstrained bytes referenced from 0x4010fd
           (foo+0x4 in mem-demo.so (0x10fd))
WARNING  | Filling register 68 with 4 unconstrained bytes referenced from 0x401107
           (foo+0xe in mem-demo.so (0x1107))
```

in angr (0b)

```
p = angr.Project('mem-demo.so', load_options={'auto_load_libs': False})
x = claripy.BVS('x', 31).zero_extend(1)
y = claripy.BVS('y', 31).zero_extend(1)
foo_func = p.loader.find_symbol('foo').rebased_addr
foo_prototype = angr.sim_type.SimTypeFunction(
    args=(angr.sim_type.SimTypeInt(), angr.sim_type.SimTypeInt()),
    returnty=angr.sim_type.SimTypeInt()
)
st = p.factory.call_state(foo_func, x, y, prototype=foo_prototype,
    add_options=[
        angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
        angr.options.SYMBOL_FILL_UNCONSTRAINED_REGISTERS,
    ]
)
st.add_constraints(x < 2, y < 2)
sm = p.factory.simulation_manager(st, save_unconstrained=True)
sm.explore()
print(sm)
    # <SimulationManager with 1 deadended>
print(sm.deadended[0].regs.eax)
    # <BV32 0x40>
```

what's angr doing?

```
...
print(sm.deadended[0].solver.min(x))
    # 1
print(sm.deadended[0].solver.max(x))
    # 1
print(sm.deadended[0].solver.min(y))
    # 1
print(sm.deadended[0].solver.max(y))
    # 1
```

problem: angr's default memory write strategy: use maximum possible value for address

compromise:
often close enough
simplifies work for equation solver dramatically
often allows avoiding symbolic execution for much faster emulation

requesting symbolic addressing (1)

```
st = p.factory.call_state(foo_func, x, y, prototype=foo_prototype,
    add_options=[
        angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
        angr.options.SYMBOL_FILL_UNCONSTRAINED_REGISTERS,
        angr.options.SYMBOLIC_WRITE_ADDRESSES, # ****** ADDED ******
    ]
)
...
print(sm.deadended[0].regs.eax)
```

```
<BV32 0x20 + (if x_0_31 == 0x1 then (if y_1_31 == 0x1 then 0x20 else
(if x_0_31 == 0x1 then 0x10 else 0x2)) else (if x_0_31 == 0x0 then
(if y_1_31 == 0x0 then 0x20 else (if x_0_31 == 0x0 then 0x10 else 0x1))
else 0xc0deb4be))>
```

requesting symbolic addressing (2)

```
<BV32 0x20 + (if x_0_31 == 0x1 then (if y_1_31 == 0x1 then 0x20 else
(if x_0_31 == 0x1 then 0x10 else 0x2)) else (if x_0_31 == 0x0 then
(if y_1_31 == 0x0 then 0x20 else (if x_0_31 == 0x0 then 0x10 else 0x1))
else 0xc0deb4be))>
```

0x20 +

- if x==1:
  - if y == 1: 0x20
  - if x == 1: 0x10
  - else [impossible]: 0x2 (original array value)
- if x==0:
  - if y == 0: 0x20
  - if x == 0: 0x10
  - else [impossible]: 0x1 (original array value)

with more possibilities

```
int array[30] = { ... }
...
st.add_constraints(x < 30, y < 30) # instead of x < 2, y < 2
...
```

result: much bigger expression...

even more possibilities

```
...
st.add_constraints(x < 100, y < 100) # instead of x < 2, y < 2
...
```

problem: angr doesn't consider all possibilities for write by default if more than 128 bytes worth

so: select a new set of strategies

```
st.memory.write_strategies = [
    angr.concretization_strategies.SimConcretizationStrategyRange(4 * 100),
    angr.concretization_strategies.SimConcretizationStrategyMax()
]
st.memory.read_strategies = ... # similar
```

when accessing memory, consider results of:

a range of up 400 bytes of possibilities *OR*

the maximum possible value

speed changes (with slow install of angr)

just loading file + setting up: about 1.6 s

$x < 2$, $y < 2$: around 2.0 s (0.4 s of work)

$x < 30$, $y < 30$: around 2.2 s (0.6 s of work)

$x < 100$, $y < 100$: around 2.6 s (1.0 s of work)

$x < 200$, $y < 200$: around 4.3 s (2.9 s of work)

$x < 300$, $y < 300$: around 6.5 s (4.9 s of work)

$x < 400$, $y < 400$: around 9.4 s (7.8 s of work)

handling I/O

so far: showed function arguments being variables

often: want to handle I/O

idea: track contents of files like arrays in memory

- bytes of file can be symbolic (expressions)
- read/write operations use/modify these arrays

want to test different inputs

- initialize virtual file to fixed contents

angr: symbolic input

```
# array of 5000 symbolic bytes named in_0, in_1, ...
input_chars = [claripy.BVS('in_{}'.format(i), 8) for i in range(5000)]
the_input = claripy.Concat(*input_chars)

# setup symbolic execution state with stdin being this file
st = p.factory.full_init_state(...,
    stdin=angr.SimFileStream(name='stdin', content=the_input, has_end=True),
)
...
```

simulation runs copying symbolic values when stdin is read

then, find a specific state, solve for ***a*** possible value

(0 = Linux 'file descriptor' for stdin; 1 = stdout)

```
print("stdin", some_state.posix.dumps(0))
print("stdout", some_state.posix.dumps(1))
```

angr and virtual files

angr: two ways to intercept file operations

one: intercept system calls (requests to OS)

- means very Linux-specific and very Windows-specific code
- one gets()/fgets()/etc. operation often turns into many system calls

two: replace libc functions with 'symbolic' versions

- faster
- doesn't work in statically linked programs
- won't detect issues resulting from libc details
- (example: leftover uninitalized values)

example: other libc interception

```
class memcpy(angr.SimProcedure):
  def run(self, dst_addr, src_addr, limit):
    if not self.state.solver.symbolic(limit):
        conditional_size = self.state.solver.eval(limit)
    else:
      # constraints on the limit are added during the store
      max_memcpy_size = self.state.libc.max_memcpy_size
      max_limit = self.state.solver.max_int(limit)
      min_limit = self.state.solver.min_int(limit)
      conditional_size = min(max_memcpy_size, max(min_limit, max_limit))
      ...
    ...
    # simplified
    src_mem = self.state.memory.load(src_addr, conditional_size, endness="Iend_BE"
    self.state.memory.store(dst_addr, src_mem, size=conditional_size, endness="Ier

    return dst_addr
```

approximation — choosing specific size

angr and ‘solving’ overflows/etc. (1)

```
...
p = angr.Project('./over.exe', load_options={'auto_load_libs': False})
input_chars = [claripy.BVS('in_{}'.format(i), 8) for i in range(5000)]
the_input = claripy.Concat(*input_chars)
st = p.factory.full_init_state(...,
    stdin=angr.SimFileStream(name='stdin', content=the_input, has_end=True),
)
st.libc.max_gets_size = 4500
sm = p.factory.simulation_manager(st, save_unconstrained=True)
```

angr and ‘solving’ overflows/etc. (2)

```
...
sm = p.factory.simulation_manager(st, save_unconstrained=True)
while True:
    sm.step()
    for un_st in sm.unconstrained:
        if un_st.solver.symbolic(un_st.regs_pc):
            un_st.add_constraints(un_st.regs.pc == 0x12345678)
            print(un_st.posix.dumps(0)) # compute stdin value
            break
```

my desktop w/ slow angr install: produces input that overwrites return address 0x12345678 for OVER in about 2 minutes

semi-automatic exploits?

finding what part of input sets PC to particular value is good start

can extend further:

scan emulated environment for executable memory with symbolic contents

- angr: enable lookup table (input byte $\rightarrow$ where in simulated memory)

check how value of registers/stack pointer relates to input

- see which registers/etc. are symbolic
- ask equation solver to set to particular value

symbolic execution as RE tool

angr — primarily designed for program analysis

- can tie to getting function call graph/etc. information (like displayed in Ghidra, but programmatically)

can filter symbolic execution down particular path

- identify path + solve for how to go that way
- examine registers/memory after each step
- manually identify good/bad paths

can record history of calls/memory accesses in path

for practicality: need to prune irrelevant paths

- can specify functions to avoid
- can set custom 'exploration' strategy to filter

other things we can do

intercept address concretization to ‘solve’ SUBTERFUGE, probably

reverse engineering

- find input to reach certain part of code

performance issue with symbolic exec (1)

for angr: I've been running with pypy (JIT'ing Python)

- (because easier for me to install)
- probably means I'm missing a bunch of performance

for bugfinding: something like KLEE probably better

- working from higher-level representation than assembly/machine code
- (of course, less useful for reverse engineering)

but more fundamental issues...

performance issues with symbolic exec (2)

some more fundamental issues:

- loops with variable iteration count explode number of states
- memory accesses explode number of states/condition complexity
- conditions sometimes intractable to solve

some heuristics to mitigate

- prune likely less-useful paths
- merge identical states reached in different ways
- keep symbolic inputs as small as possible
- replace uninteresting functions with approximations
- only symbolically execute 'interesting' functions
- ...

backup slides

real symbolic execution

not yet used much outside of research

old technique (1970s), but recent resurgence

- equation solving ('SAT solvers'/'SMT solvers') is now much better

example usable tools: KLEE, symcc (test case generating)

KLEE optimizations

lots of optimizations to make search time pratical

prioritize paths that produce good tests

- try to execute *new code*
- try to find new paths new root of tree

reuse equation solving results:

- remove irrelevant variables from equation solving queries
  - e.g. if (x == 10) doesn't need variables unrelated to x's value
- cache of prior queries with "no solution"

results from 1 hour of compute time (from 2008 paper):

- avg. 91% coverage on Linux coreutils (basic command line tools)
- versus developer tests: 68% covergae