Symbolic Execution

Christoph Thiede

Advisor: Patrick Rein

Advanced Programming Tools Seminar 2022/23

Software Architecture Group

Hasso Plattner Institute

Agenda

- Introduction
  - Classification
  - Definition
- Traditional symbolic execution
  - Terminology
  - Demo: KLEE
- Advanced symbolic execution
  - Challenges
  - Environment models
  - Search strategies
  - Alternative execution models (DSE, SSE, veritesting)
  - Memory models
  - Other optimizations
  - Limitations
- Impact (exploit detection)
- Tools and applications
  - Exploit detection
  - Program analysis, contract programming
    - Demo: CrossHair
  - Program exploration, unit test generation
    - Demo: Microsoft IntelliTest
  - Debugging
    - Demo: Symbolic Execution Debugger (SED)
  - Other applications
  - Impactful tools
- Conclusion

Program Analysis

Benchmarks

Contracts

Software metrics
(LOC, NOC, EOC,
LCOM, …)

Invariant inference

Dependency analysis

Program slicing

Duplicate code detection

Style checker

Tests

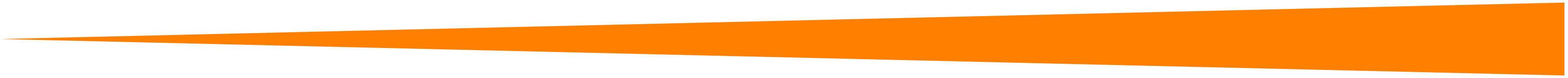
Assertions

Spell checker

Linter/bug detection

Program Analysis

amount of context



Linter/bug detection

Assertions

Duplicate code detection

Tests

Static analysis

Spell checker

Contracts

Dynamic analysis

Program slicing

Benchmarks

Dependency analysis

Invariant inference

Style checker

Software metrics
(LOC, NOC, EOC,
LCOM, …)

Program Analysis

amount of context

How can we get context?

Non-interactive argument-free commands

Tests

Symbolic execution

What is symbolic execution?

And what it is useful for (and for what not)?

What Is Symbolic Execution?

sym·bol·ic ex·e·cu·tion *n, abbrev.* **Sym·Ex** *or* **sym·bex**

- “bring your code to life”
- “turn a program inside out, so that instead of consuming inputs, it becomes a generator of them”[1]
- uncover all possible execution paths of a program

[1] [cadar2005execution]

What Is Symbolic Execution?

- execute program with symbolic values instead of concrete values for variables
- each execution path (aka symbolic state) satisfies a path constraint formula
- for each conditional branch, the execution is forked into two execution paths with divergent constraints
- all execution paths form an execution tree together
- for testing branch conditions for satisfiability, the symbolic expression is checked by an SMT solver
  - SMT = satisfiability modulo (“within”) theories
  - special form of SAT solver

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
           a = 0  b = 2  c = -1
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

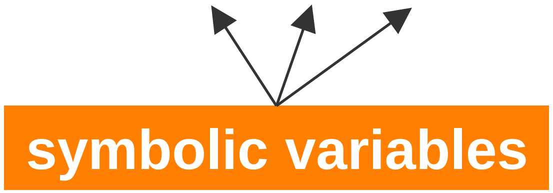
- `foo(1, -1, 2)`
- `foo(0, 2, -1)`
- `foo(0, -1, 2)`

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
         a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

- foo(1, -1, 2)
- foo(0, 2, -1)
- foo(0, -1, 2)
- foo(α, β, γ)



symbolic variables

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
                a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
                a = α  b = β  c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
    a = α   b = β   c = γ         1
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α != 0
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
                a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α = 0
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

1 2

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
         a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α != 0
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

1 2

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

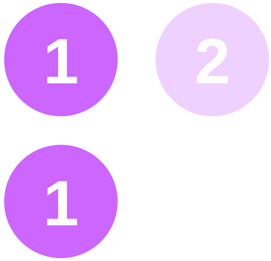
1 2

```
         a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;   x = -2
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

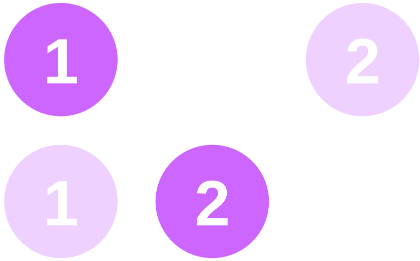
```
             a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α != 0
        x = -2;   x = -2
    }
    if (b < 0) {  β < 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

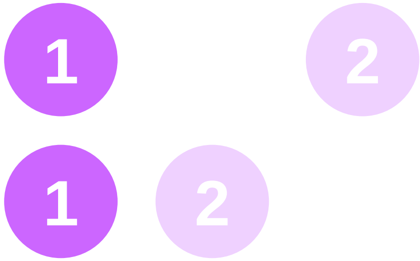
```
             a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β >= 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

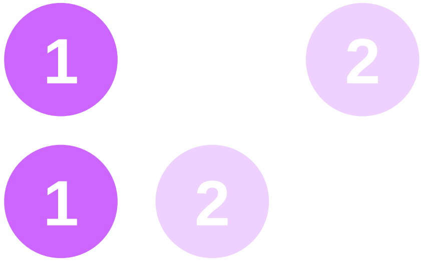
```
         a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

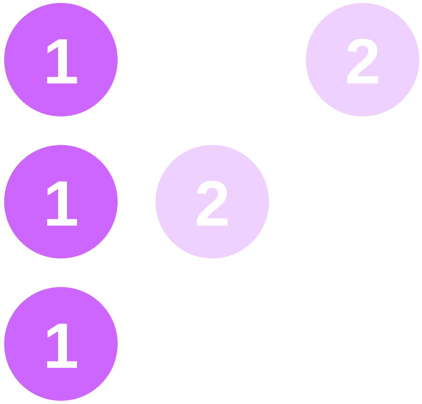
```
             a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {    α != 0
        x = -2;     x = -2
    }
    if (b < 0) {    β < 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

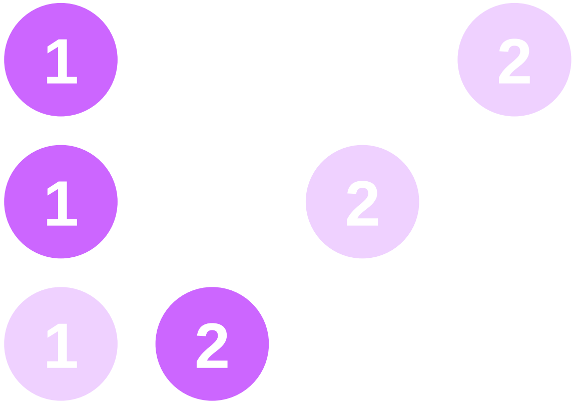
```
         a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

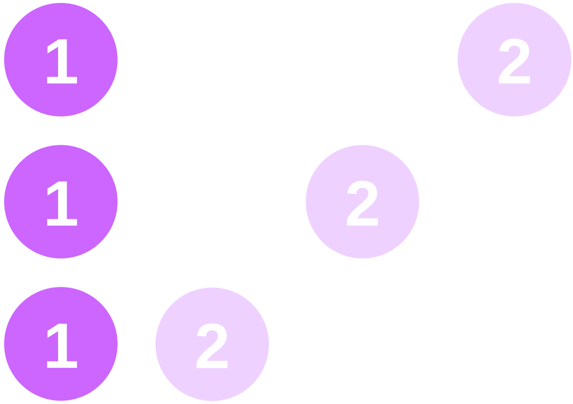
```
         a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α != 0
        x = -2;   x = -2
    }
    if (b < 0) {  β < 0
        if (!a && c) {  γ = 0
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

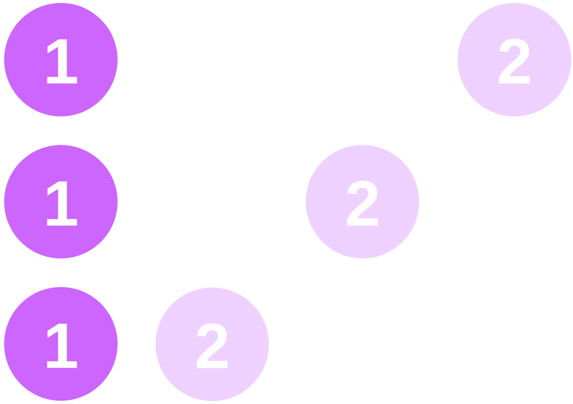
```
         a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {    α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

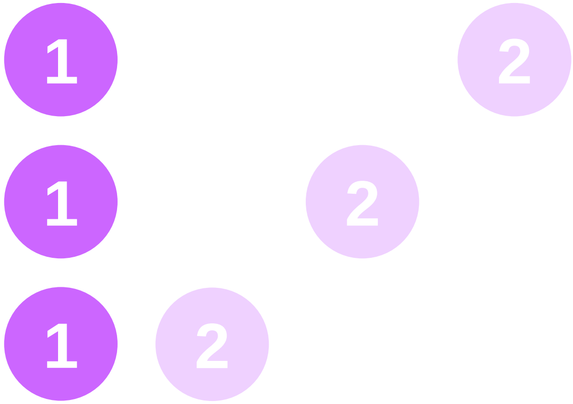
```
                a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;    y = 1
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

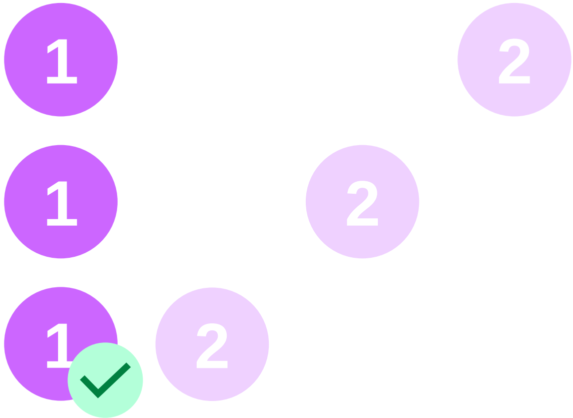
```
          a = α      b = β      c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;    y = 1
        }
        z = 2;    z = 2
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

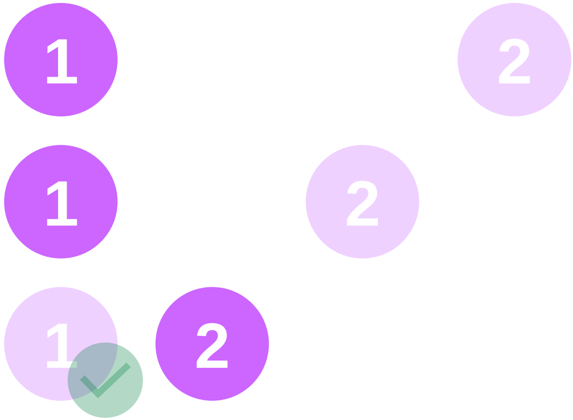
```
          a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;     x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;    y = 1
        }
        z = 2;    z = 2
    }
    assert(x + y + z != 3);    1 != 3    ✓
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

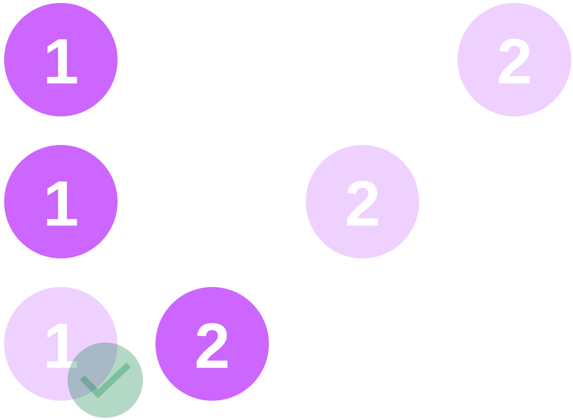
```
             a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {    α != 0
        x = -2;     x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ = 0
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

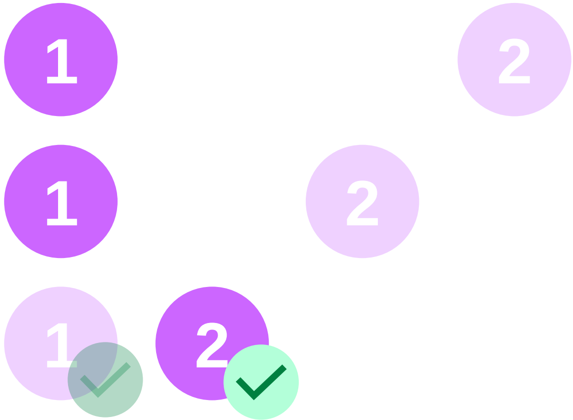
```
         a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ = 0
            y = 1;
        }
        z = 2;    z = 2
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

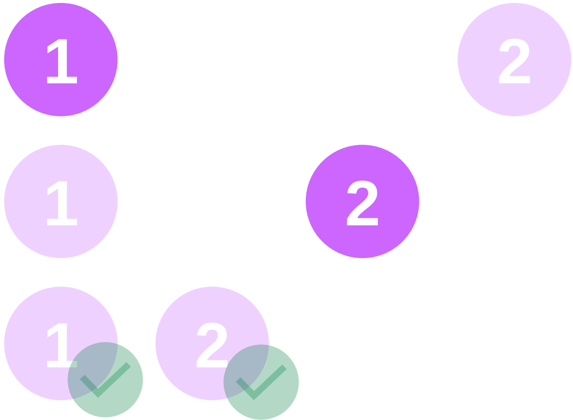
```
         a = α     b = β     c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ = 0
            y = 1;
        }
        z = 2;    z = 2
    }
    assert(x + y + z != 3);    0 != 3  ✓
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

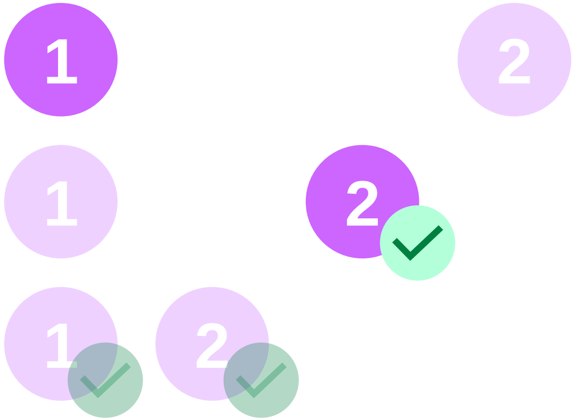
```
                 a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;     x = -2
    }
    if (b < 0) {   β >= 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

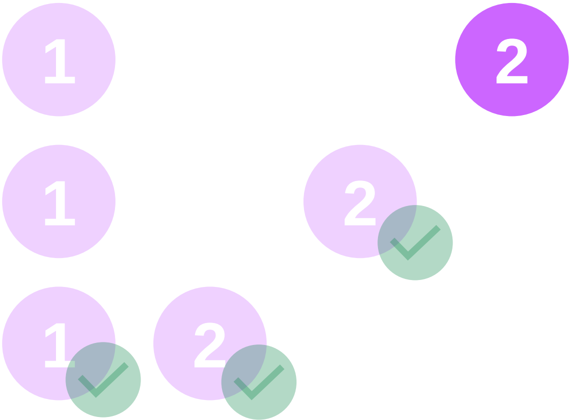
```
          a = α     b = β     c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α != 0
        x = -2;    x = -2
    }
    if (b < 0) {   β >= 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);    -2 != 3   ✓
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

```
         a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α = 0
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

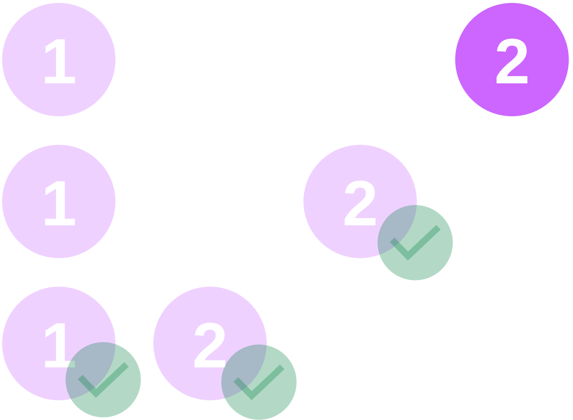


[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

a = α   b = β   c = γ

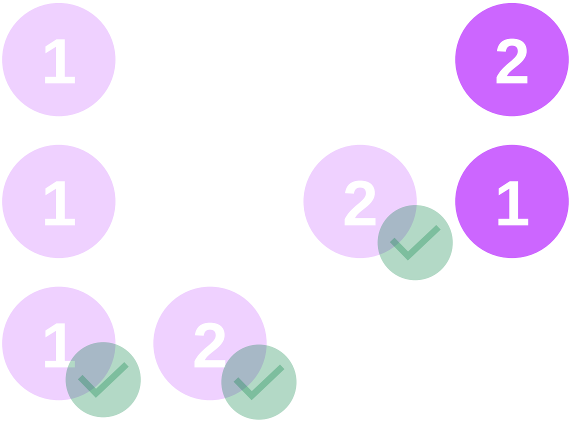
```
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α = 0
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

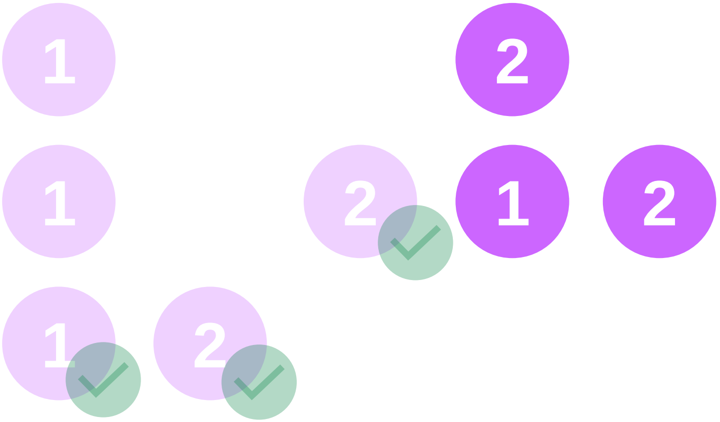
```
                  a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α = 0
        x = -2;
    }
    if (b < 0) {  β < 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

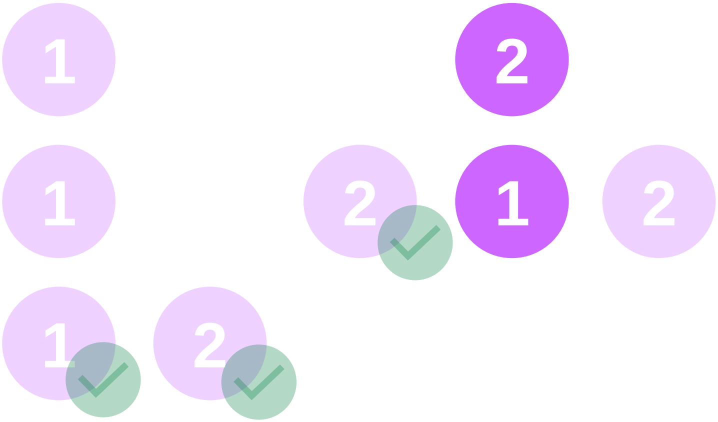
```
          a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α = 0
        x = -2;
    }
    if (b < 0) {  β >= 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

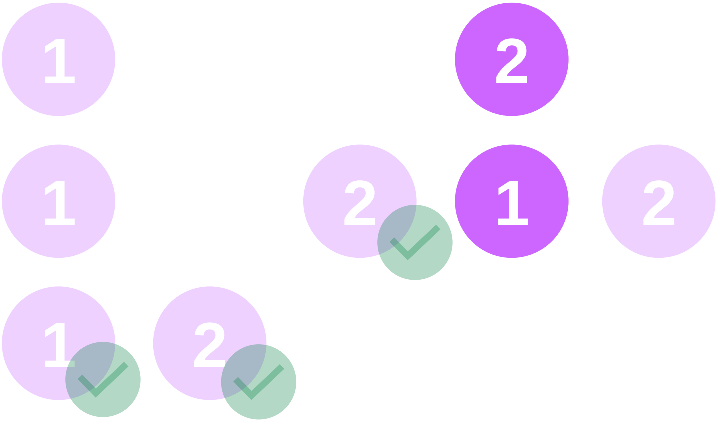
```
             a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α = 0
        x = -2;
    }
    if (b < 0) {  β < 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

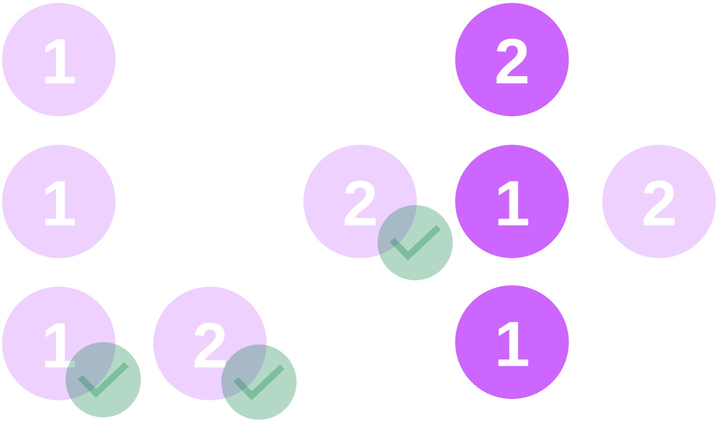
```
               a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α = 0
        x = -2;
    }
    if (b < 0) {   β < 0
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

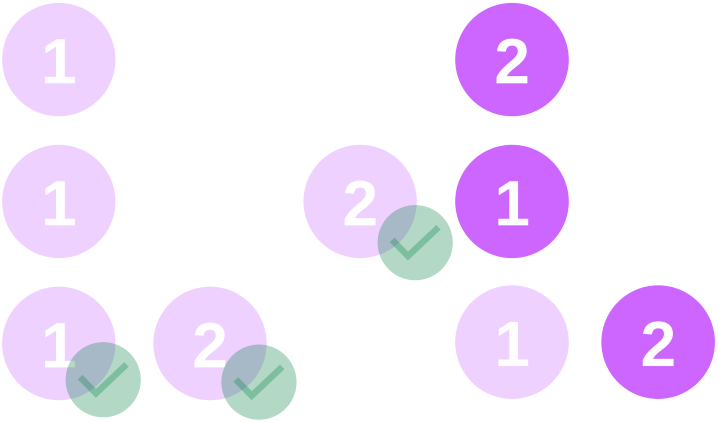
```
             a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α = 0
        x = -2;
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

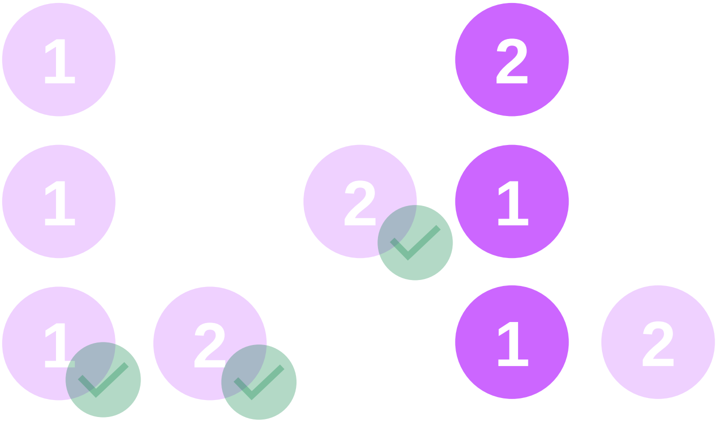
```
              a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {  α = 0
        x = -2;
    }
    if (b < 0) {  β < 0
        if (!a && c) {  γ = 0
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

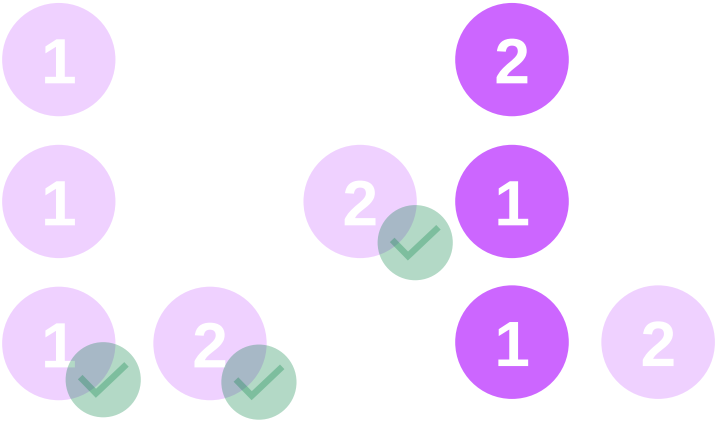
```
              a = α    b = β    c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α = 0
        x = -2;
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

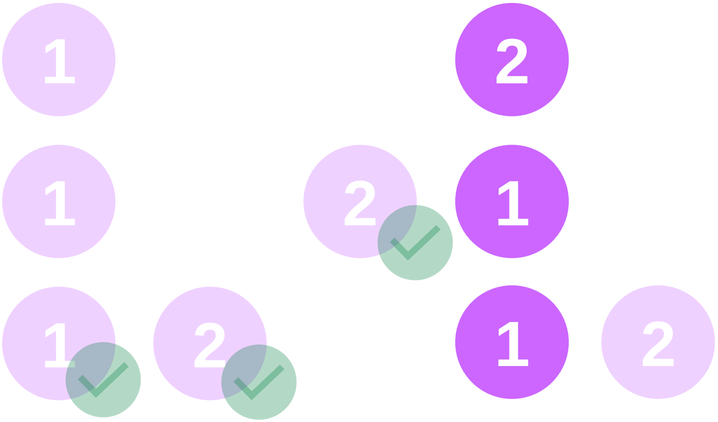
```
              a = α     b = β     c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {    α = 0
        x = -2;
    }
    if (b < 0) {    β < 0
        if (!a && c) {    γ != 0
            y = 1;    y = 1
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?

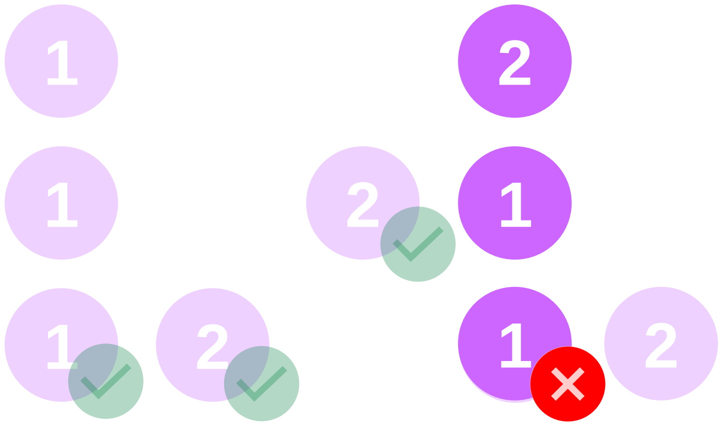
```
         a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α = 0
        x = -2;
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;   y = 1
        }
        z = 2;   z = 2
    }
    assert(x + y + z != 3);
}
```



[cadar2013symbolic, baldoni2018survey]

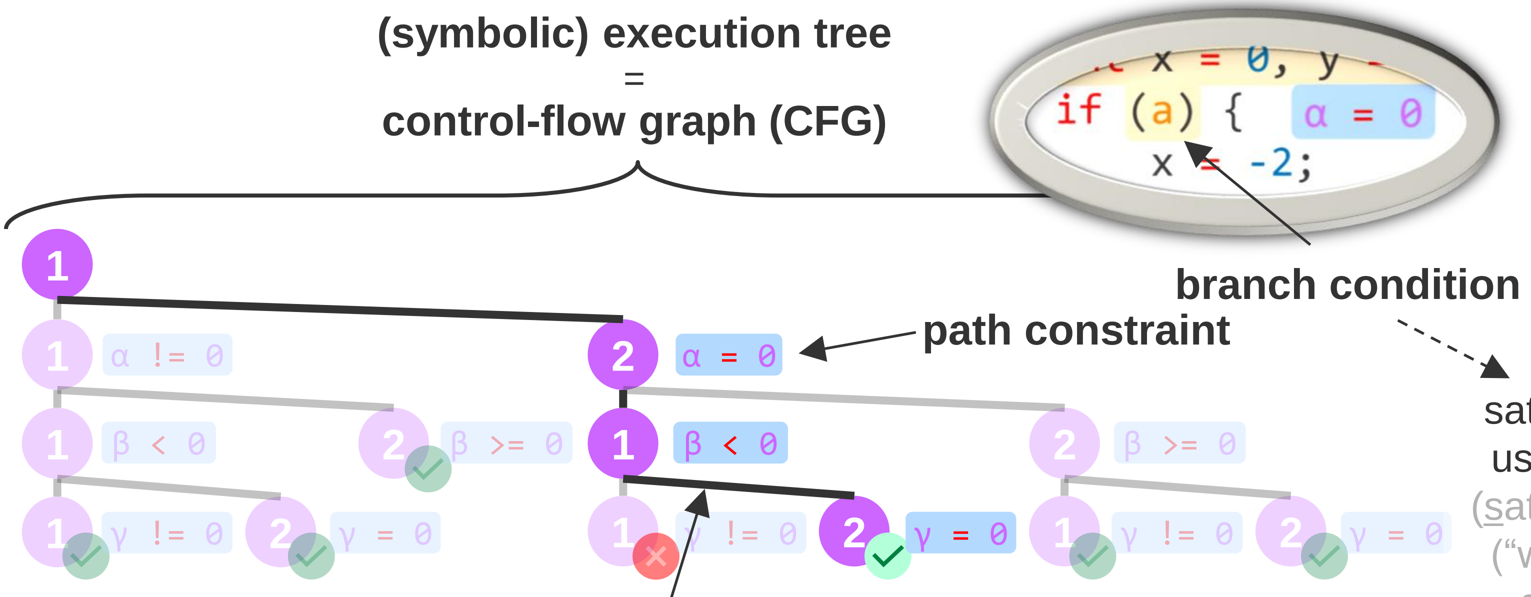
What Is Symbolic Execution?

```
         a = α   b = β   c = γ
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {   α = 0
        x = -2;
    }
    if (b < 0) {   β < 0
        if (!a && c) {   γ != 0
            y = 1;   y = 1
        }
        z = 2;   z = 2
    }
    assert(x + y + z != 3);   3 = 3   ✗
}
```



[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?



(symbolic) execution tree
=
control-flow graph (CFG)

branch condition

path constraint

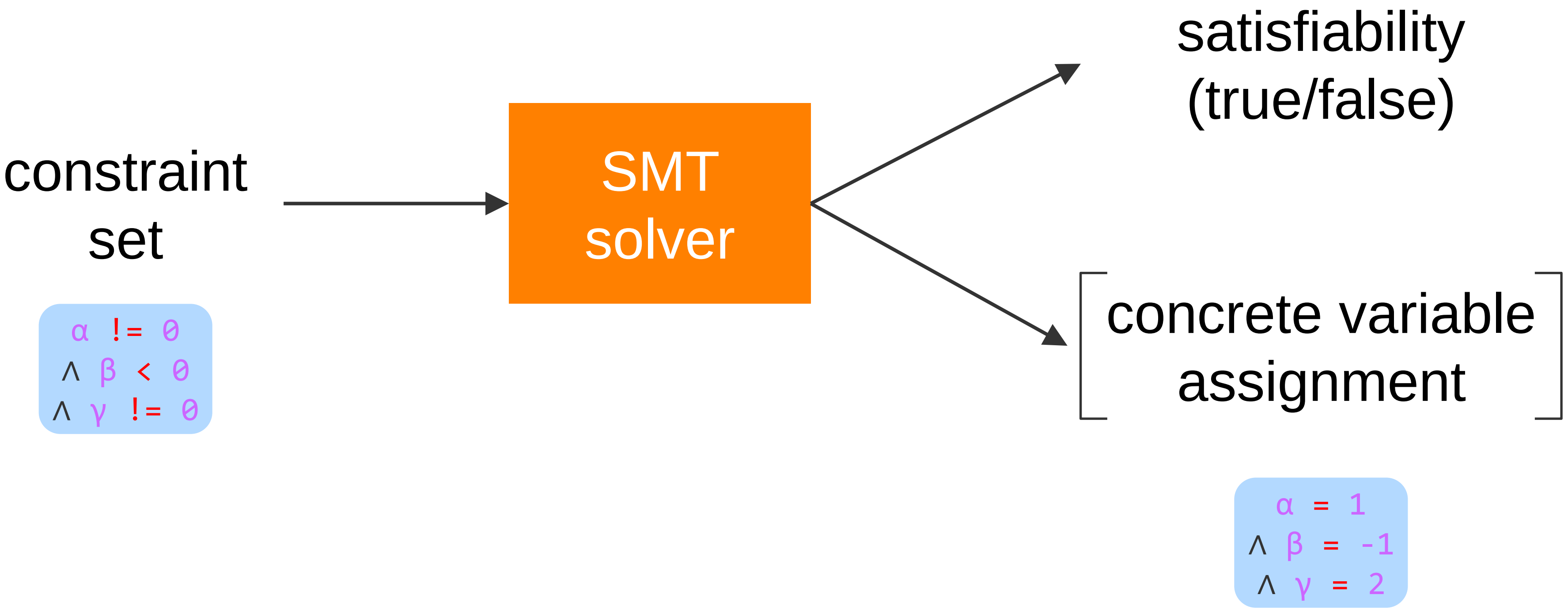
(symbolic) execution path

constraint set
=
symbolic state

satisfiability testing
using **SMT solver**
(satisfiability modulo
("within") theories,
special form of
SAT solver)

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?



constraint set

```
α != 0
∧ β < 0
∧ γ != 0
```

SMT solver

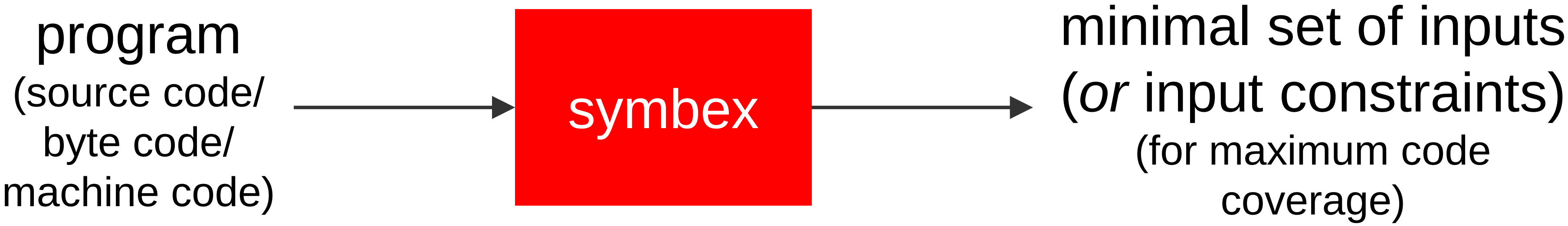
satisfiability (true/false)

concrete variable assignment

```
α = 1
∧ β = -1
∧ γ = 2
```

[cadar2013symbolic, baldoni2018survey]

What Is Symbolic Execution?



program
(source code/
byte code/
machine code)

symbex

minimal set of inputs
(*or* input constraints)
(for maximum code
coverage)

```
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

- `foo(1, -1, 2)`
- `foo(0, 2, -1)`
- `foo(0, -1, 2)`
- …

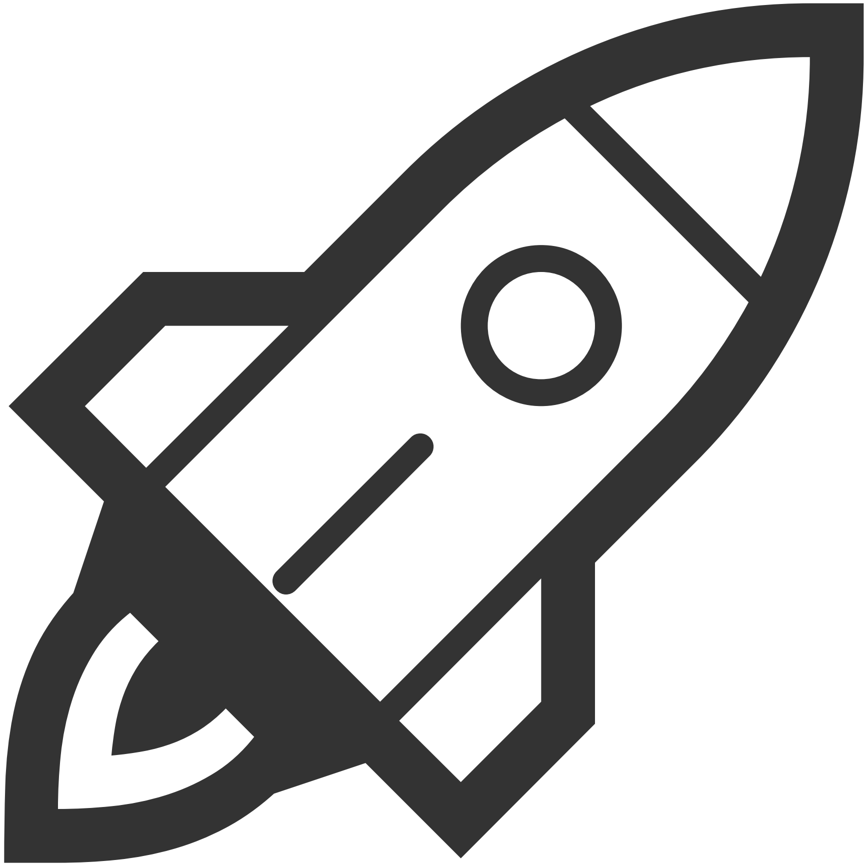
α != 0 ∧ β < 0 ∧ γ != 0

α = 0 ∧ β < 0 ∧ γ = 0

…

[cadar2013symbolic, baldoni2018survey]

Demo: KLEE



[cadar2008klee]

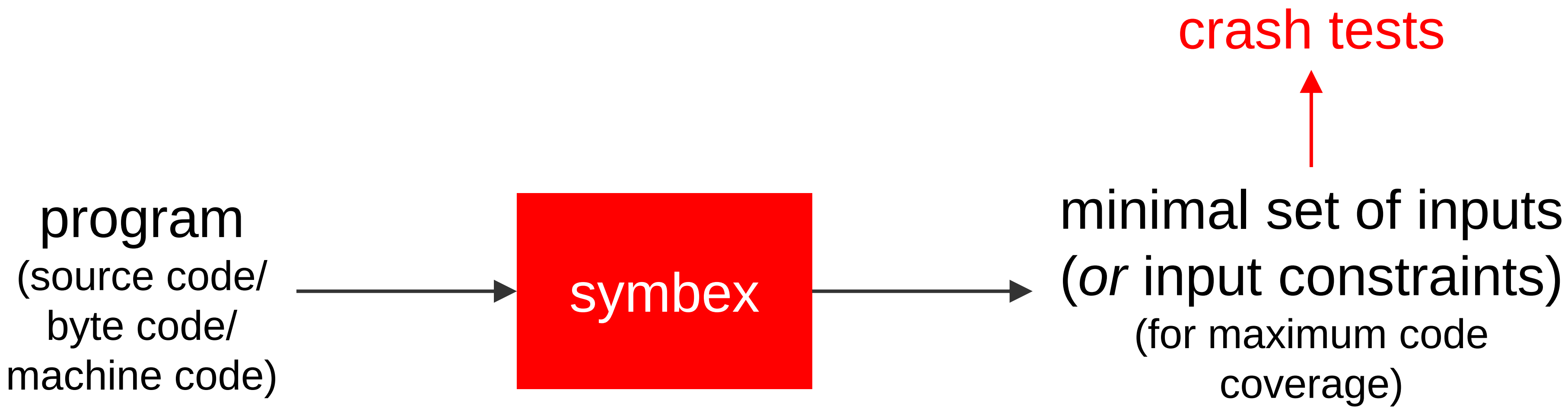
Demo: KLEE

```
$ cat >> foo.c
int main() {
    int a, b, c;
    klee_make_symbolic(&a, sizeof(a), "a");
    klee_make_symbolic(&b, sizeof(b), "b");
    klee_make_symbolic(&c, sizeof(c), "c");
    foo(a, b, c);
    return 0;
}
$ clang -emit-llvm -c foo.c
$ klee foo.bc
KLEE: output directory is "/home/andrew/klee-out-0"
KLEE: Using STP solver backend
KLEE: ERROR: (location information missing) abort
failure
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 99
KLEE: done: completed paths = 4
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 5
$ ls klee-last
assembly.ll  messages.txt  run.stats
test000002.ktest  test000004.ktest
test000005.kquery  warnings.txt
info          run.istats    test000001.ktest
test000003.ktest  test000005.abort.err
test000005.ktest
```

```
$ cat klee-last/test000005.abort.err
Error: abort failure
Stack:
        #000000066 in foo(a=symbolic, b=symbolic,
c=symbolic)
        #100000093 in main()
$ ktest-tool klee-last/test000005.ktest
ktest file : 'klee-last/test000005.ktest'
args       : ['foo.bc']
num objects: 3
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
object 1: name: 'b'
object 1: size: 4
object 1: data: b'\x00\x00\x00\x80'
object 1: hex : 0x00000080
object 1: int : -2147483648
object 1: uint: 2147483648
object 1: text: ....
object 2: name: 'c'
object 2: size: 4
object 2: data: b'\x01\x01\x01\x01'
object 2: hex : 0x01010101
object 2: int : 16843009
object 2: uint: 16843009
object 2: text: ....
```

[cadar2008klee]

What Is Symbolic Execution?



crash tests

program
(source code/
byte code/
machine code)

symbex

minimal set of inputs
(*or* input constraints)
(for maximum code
coverage)

```
void foo(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) {
        x = -2;
    }
    if (b < 0) {
        if (!a && c) {
            y = 1;
        }
        z = 2;
    }
    assert(x + y + z != 3);
}
```

- `foo(1, -1, 2)`
- `foo(0, 2, -1)`
- `foo(0, -1, 2)`
- …

`α != 0 ∧ β < 0 ∧ γ != 0`

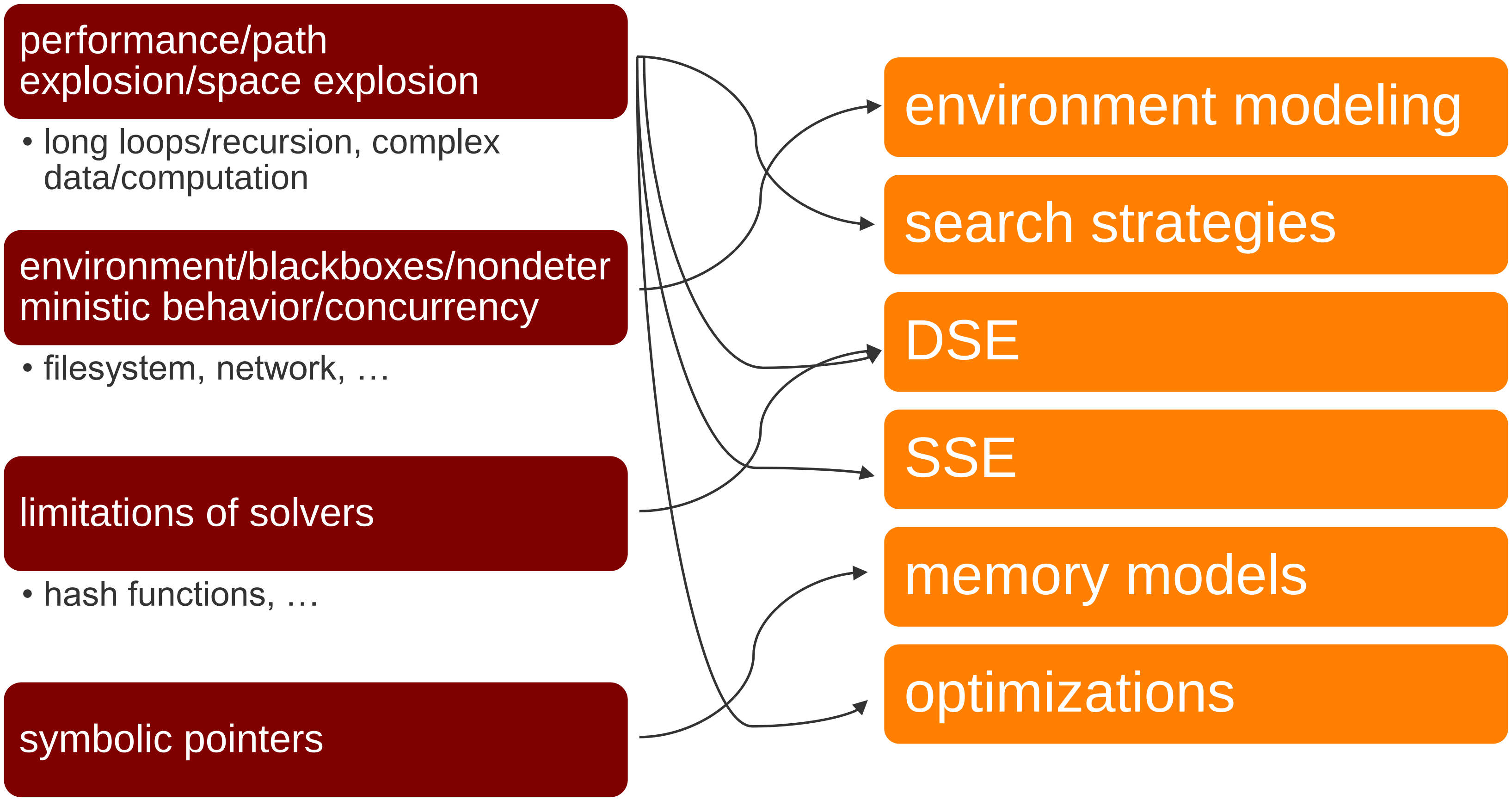
`α = 0 ∧ β < 0 ∧ γ = 0`

…

[cadar2008klee]

What could possibly go wrong?

Challenges and Solutions



performance/path explosion/space explosion

- long loops/recursion, complex data/computation

environment/blackboxes/nondeterministic behavior/concurrency

- filesystem, network, …

limitations of solvers

- hash functions, …

symbolic pointers

environment modeling

search strategies

DSE

SSE

memory models

optimizations

[cadar2013symbolic, baldoni2018survey]

Unknown Environment

- environment = unknown behavior (blackboxes)
  - not symbolically executable
  - syscalls (file system, network, user input, hardware acceleration, …)

```
int foo(char *fname) {
    FILE *f = fopen(fname, "r");
    if (f == NULL) {
      return -1;
    }
    // ...
    fclose(f);
    return 0;
}
```

  - frameworks without sources (only for symbolic interpreters)
  - concurrency (scheduler)

[cadar2013symbolic, baldoni2018survey, yang2019advances]

Environment Modeling

- stubs
- mocks

```
class VirtualFileSystem {…}
```

- drivers

```
requests.get = MagicMock(
    return_value=['first', 'second', 'third'])
```

- heuristics
  - only model problematic subset of behavior (e.g., exceptions)
  - frameworks (inversion of control): connect sub-call graphs using regular grammar

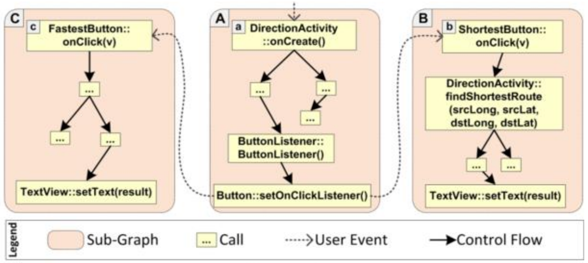

C
c FastestButton:: onClick(v)
...
...
...
TextView::setText(result)
A
a DirectionActivity ::onCreate()
...
...
...
ButtonListener:: ButtonListener()
Button::setOnClickListener()
B
b ShortestButton:: onClick(v)
DirectionActivity:: findShortestRoute (srcLong, srcLat, dstLong, dstLat)
...
...
TextView::setText(result)
Legend
Sub-Graph
... Call
User Event
Control Flow

Figure 3. a subset of the call graph model for *Driving Directions* app.

(1) $S \rightarrow aA$

(2) $A \rightarrow bB|cC|\varepsilon$

(3) $B \rightarrow A|\varepsilon$

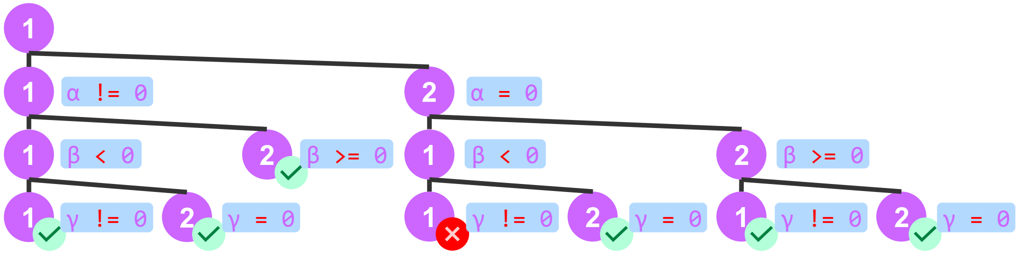
(4) $C \rightarrow A|\varepsilon$

- trade-offs
  - provided by symbex engine vs extended by users
  - model completeness vs complexity/performance

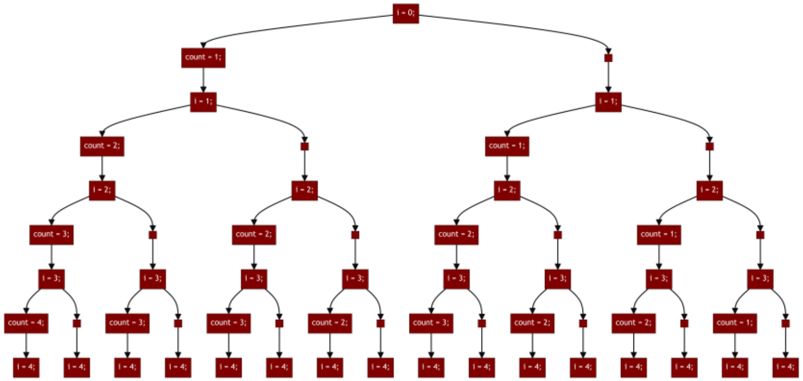
[cadar2013symbolic, baldoni2018survey, mirzaei2012testing, yang2019advances]

Path Explosion

- number of execution paths = 2^(number of branches)



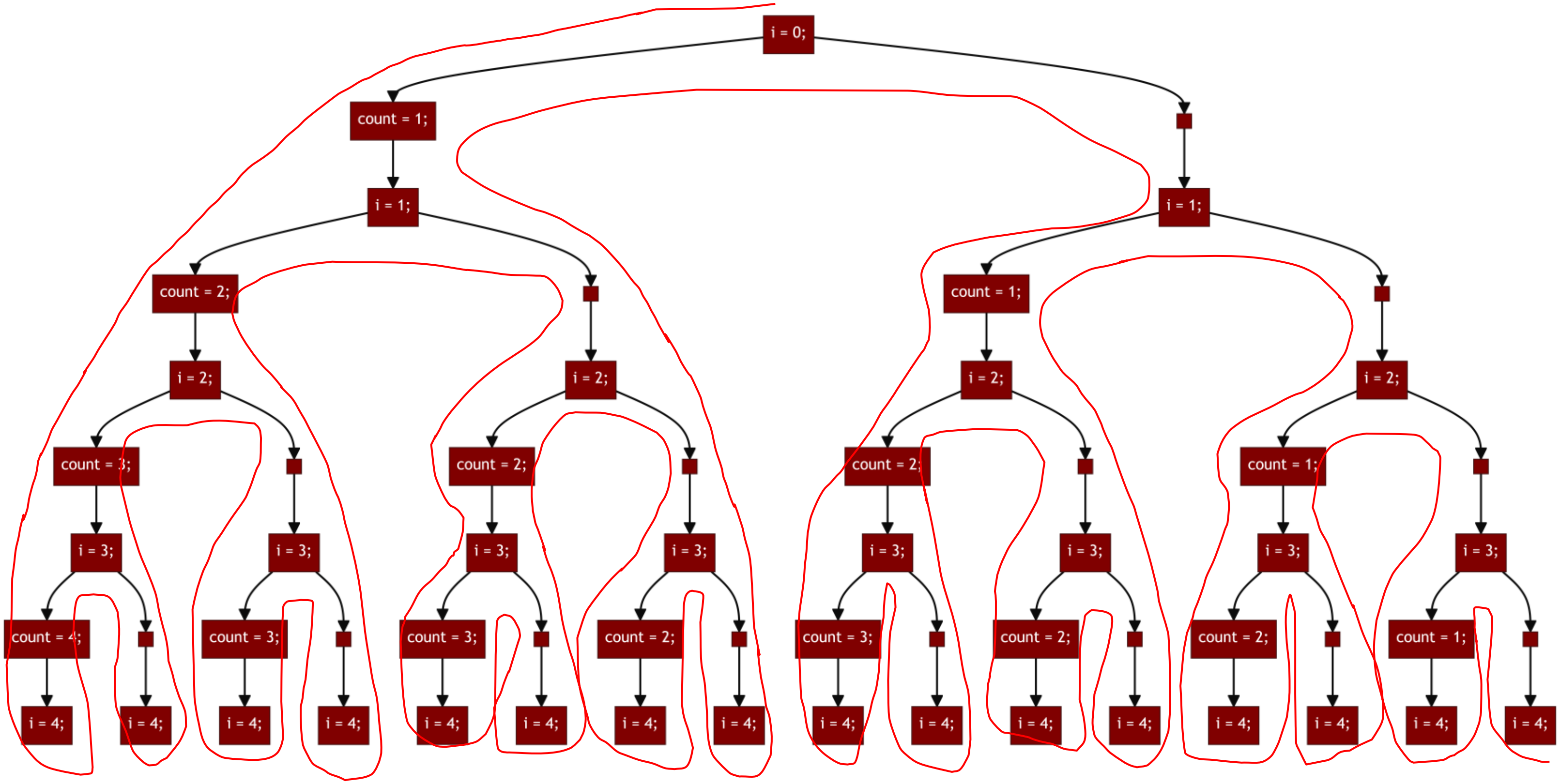
```
int count(int *array, int value) {
    int i, count = 0;
    for (i = 0; i < 100; i++) {
            if (array[i] == value) {
                count++;
            }
    }
    return count;
}
```



- infinite tree for conditional loops

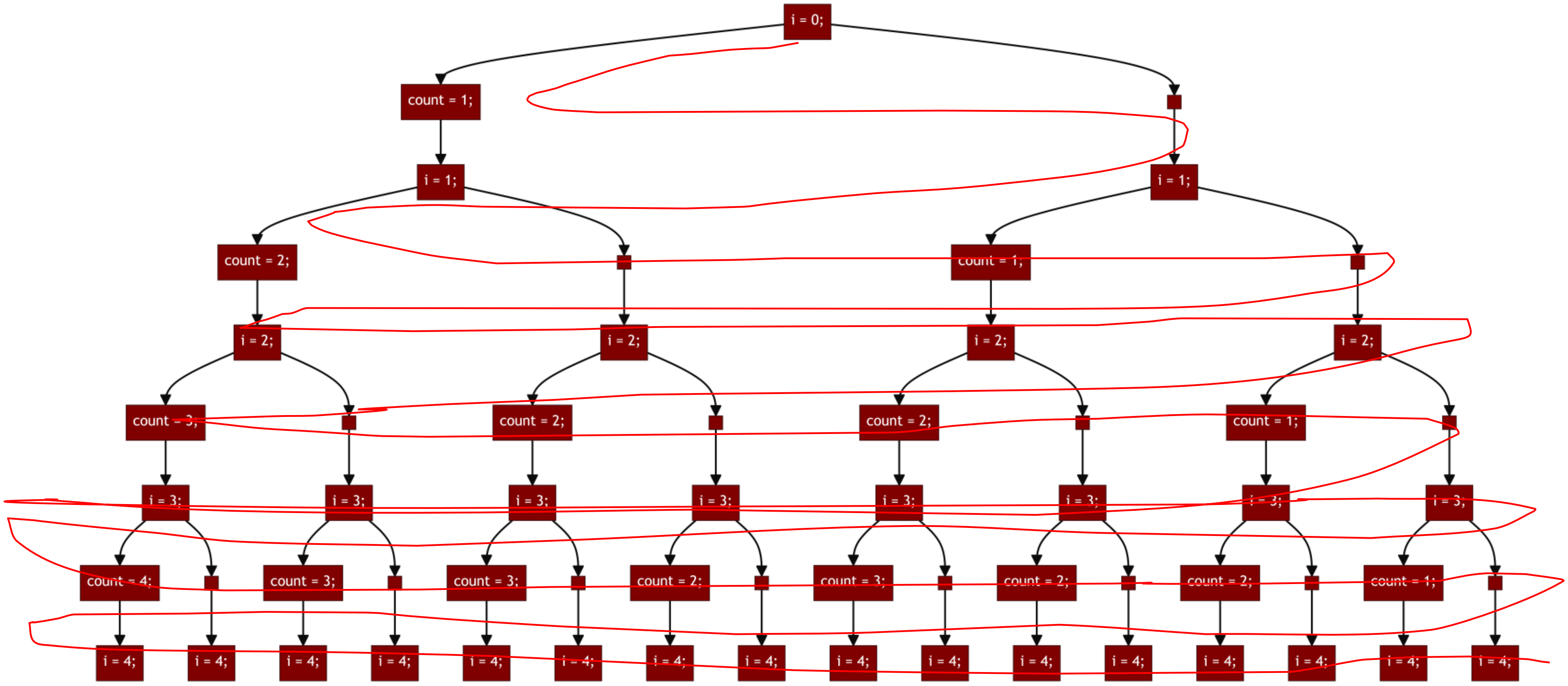
[cadar2013symbolic, baldoni2018survey, yang2019advances]

Path Explosion (DFS)



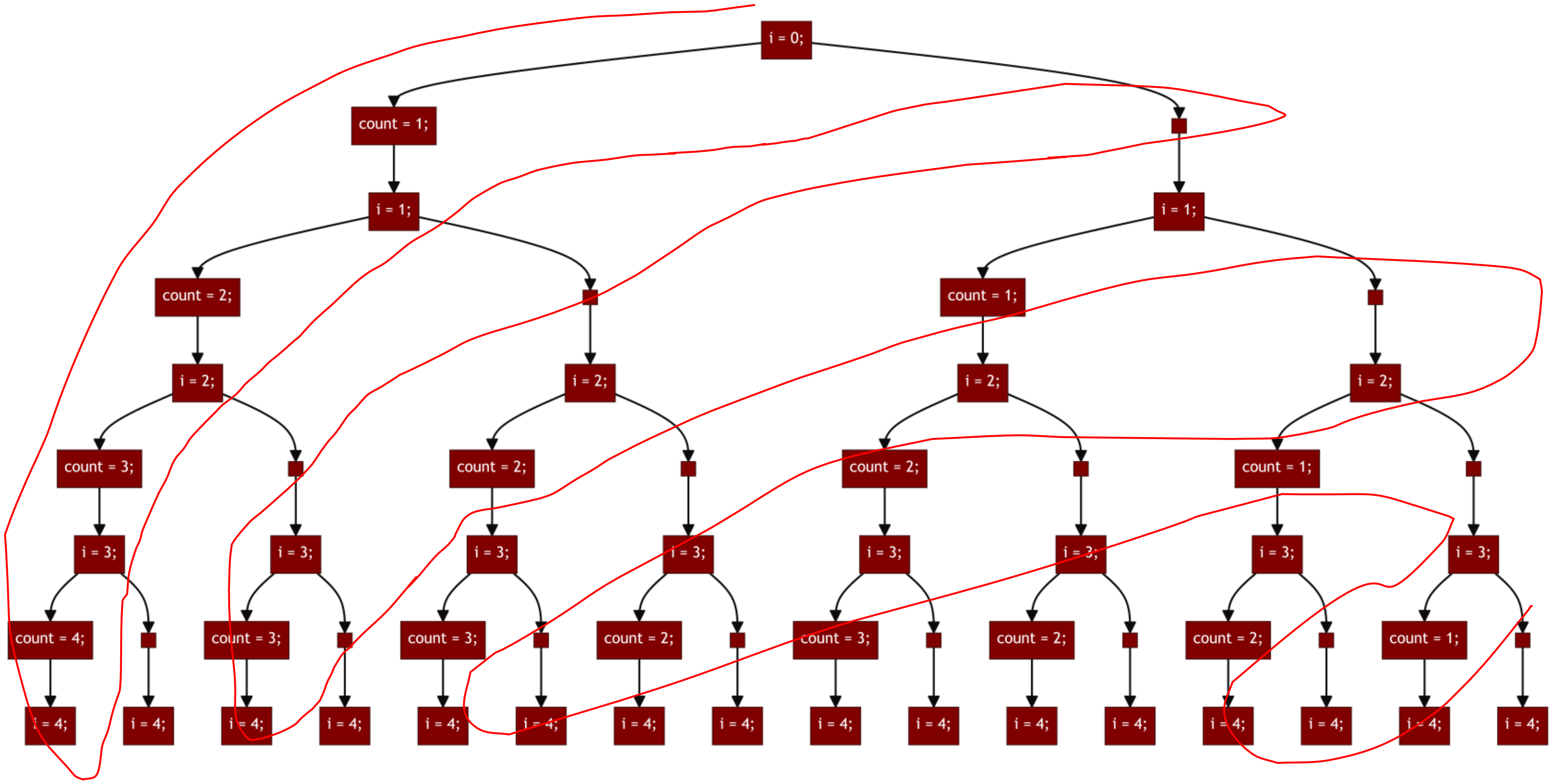
[godefroid2008automated, cadar2013symbolic, liu2017survey, baldoni2018survey, yang2019advances]

Path Explosion (BFS)



[godefroid2008automated, cadar2013symbolic, liu2017survey, baldoni2018survey, yang2019advances]

Path Explosion (generational search)



[godefroid2008automated, cadar2013symbolic, liu2017survey, baldoni2018survey, yang2019advances]

Path Explosion

- search strategies
  - depth-first search (DFS): not loop-safe
  - bread-first search (BFS): many context switches, memory overhead
  - generational search (diagonal tree traversal): smaller memory overhead)
- branch prioritization
  - random search: not always loop-safe
  - heuristic priorities:
    - by code coverage increase
    - by control-flow graph coverage
    - by mutation coverage increase
    - evolutionary search
  - hybrid approaches (different strategies for different phases of code coverage)

[godefroid2008automated, cadar2013symbolic, liu2017survey, baldoni2018survey, yang2019advances]

Alternative Execution Models

- problems with pure symbolic execution (as assumed so far):
  - separate, slower interpreter
  - unable to deal with blackbox environments
- dynamic symbolic execution (DSE)
  - execution-generated testing (EGT)
  - concolic execution
- static symbolic execution (SSE)
  - veritesting

[cadar2013symbolic, avgerinos2014enhancing, baldoni2018survey, yang2019advances]

Dynamic Symbolic Execution (DSE)

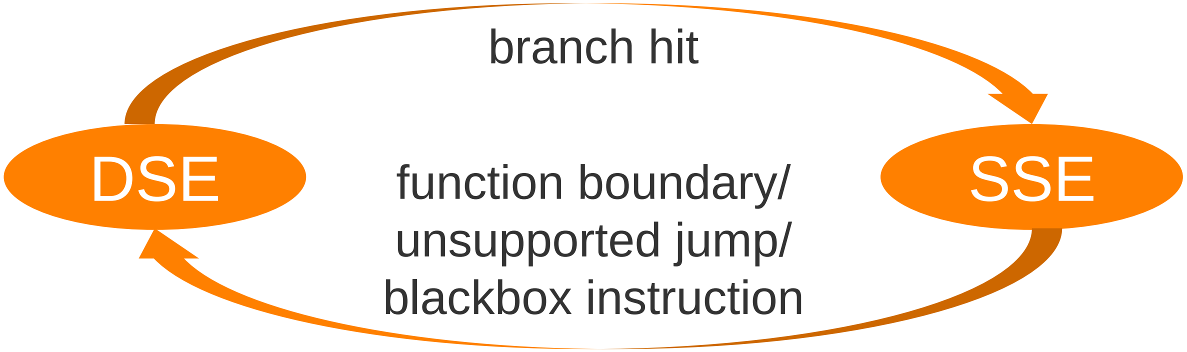
- mix concrete and symbolic execution
- execution-generated testing (EGT): interleaving mix
  - use concrete values for executing blackboxes
  - (concretized solution from SMT solver)
- concolic execution (concrete + symbolic): simultaneous mix
  - concrete execution for control flow and concrete values
  - symbolic execution for collecting constraints
  - path discovery: start from concrete tests/inputs, negate single constraint and generate new values
  - implications:
    - implementation: instead of heavy-weight symbolic interpreter, normal executor/processor can be used with instrumented program
    - undecidable branches: don't terminate execution, but omit some paths
  - state of the art technology!

[cadar2005execution, cadar2013symbolic, avgerinos2014enhancing, baldoni2018survey, yang2019advances]

Static Symbolic Execution (SSE)

- convert program source code/AST to single symbolic expression
- no overhead for executing and ordering branches, but higher pressure on SMT solver
  - efficient handling of large/infinite loops!
  - nowadays, SMT solvers are more powerful
- no support for blackboxes or (often) uncommon control flow patterns (goto, function pointers, …)

- veritesting: mix DSE and SSE



[cadar2013symbolic, avgerinos2014enhancing, baldoni2018survey, yang2019advances]

Memory Models

- Challenges
  - pointers/memory aliases/dispatching
  - symbolic memory access:

```
void divergent(int x, int y)
{
    int s[4];
    s[0] = x;
    s[1] = 0;
    s[2] = 1;
    s[3] = 2;
    if (s[x] = s[y] + 2) {
        abort(); //error
    }
}
```

- Approaches
  - limited coverage
  - precision of memory model
  - lazy initialization of pointers → reduced search space of symbolic addresses

[cadar2013symbolic, baldoni2018survey, yang2019advances]

Optimizations for Symbex

- parallelize path exploration (branch-and-bound)
- SMT solvers
  - trade-in precision (e.g., arithmetic theory vs overflow-aware theory of bitvectors)
  - incremental solving (constraint set caches)
  - reuse solutions from similar constraint sets (subsets, supersets)
  - improved performance (parallelization, HW acceleration)
- selective symbolic execution (analyze subset of program)
  - directed symbolic execution (find nearby program parts)
  - lazy test generation (top-down selection)

[cadar2013symbolic, baldoni2018survey, yang2019advances]

Trends (selection)

- Path merging and pruning
- Compositional analysis
  - analyze units separately and store pre- and postconditions
- Probabilistic symbolic execution
  - observed or developer-specified branch probabilities
  - rank exploits by probability
  - also used to predict overall performance/reliability
- Shadow symbolic execution
  - exploit symbex results for previous software version and source code diff
- Hybrid heuristics
  - exploit information from static analysis to accelerate and tune symbolic execution (branch selection, memory models, …)
  - e.g., type flow analysis
- Optimized concurrency models
- …

[kuznetsov2012efficient, cadar2013symbolic, baldoni2018survey, yang2019advances]

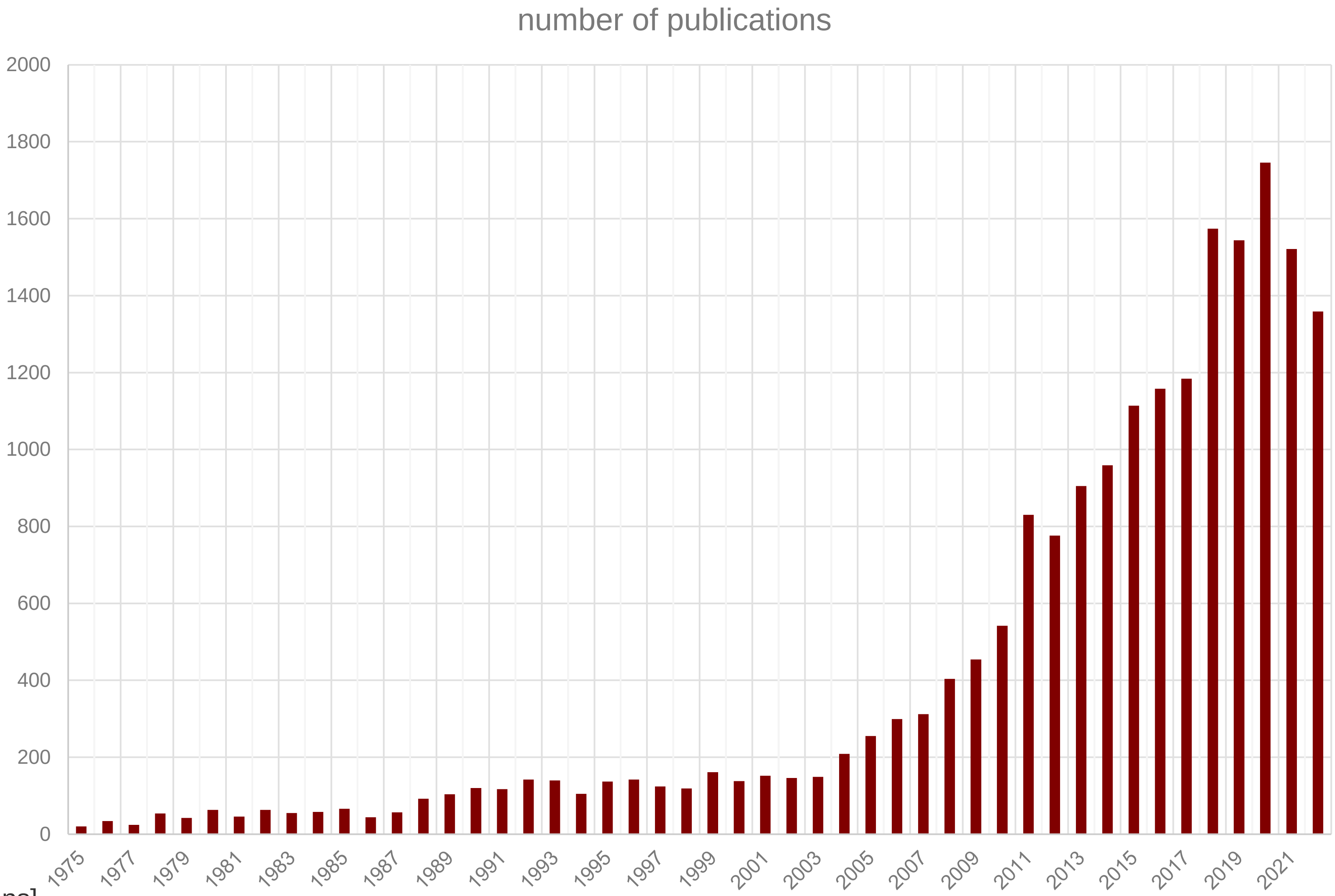
Remaining Limitations

- No 100% coverage!
  - systematic issues:
    - blackbox environments/mocking overhead
    - unsolvable constraints
    - symbolic pointers
  - performance issues (not an interactive tool)
    - execution complexity (long loops/deep recursion, many branches, large data, …)
    - computational resources

[cadar2013symbolic, baldoni2018survey, yang2019advances]

The Triumph of Symbolic Execution

The Triumph of Symbolic Execution



[dimensions]

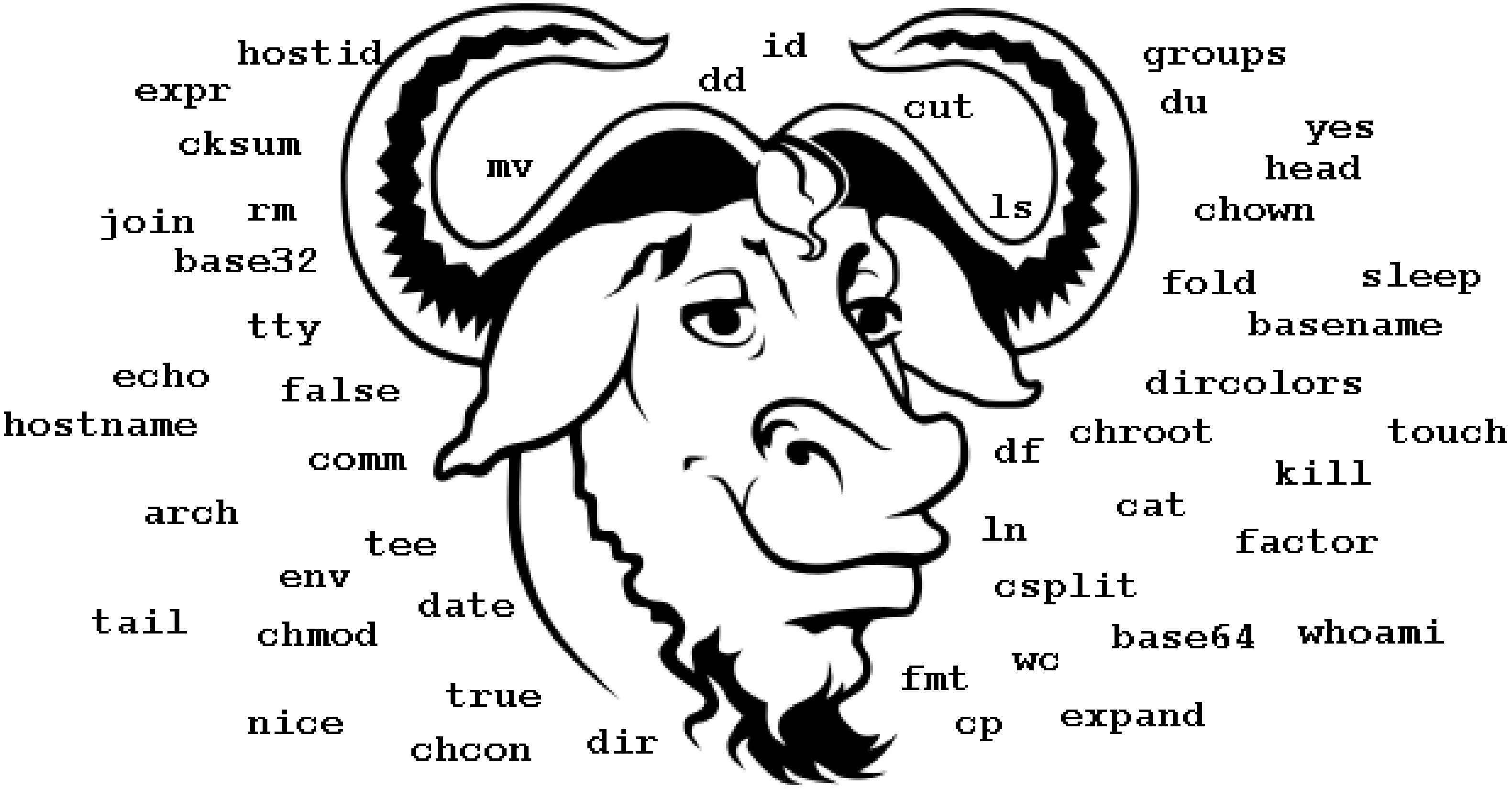
Impact (1): Microsoft SAGE

- concolic execution of x86 binaries
- “whitebox fuzzing”: find vulnerabilities in parsers by degenerating files

- responsible for finding 1/3 of exploits in Windows 7 (prior to release)
- integral part of Microsoft’s internal testing pipelines
- run 24/7 on cluster with >200 nodes

[godefroid2012sage]

Impact (2): GNU Core Utils



89 binaries · 72k SLOC · 67.6% LCOV

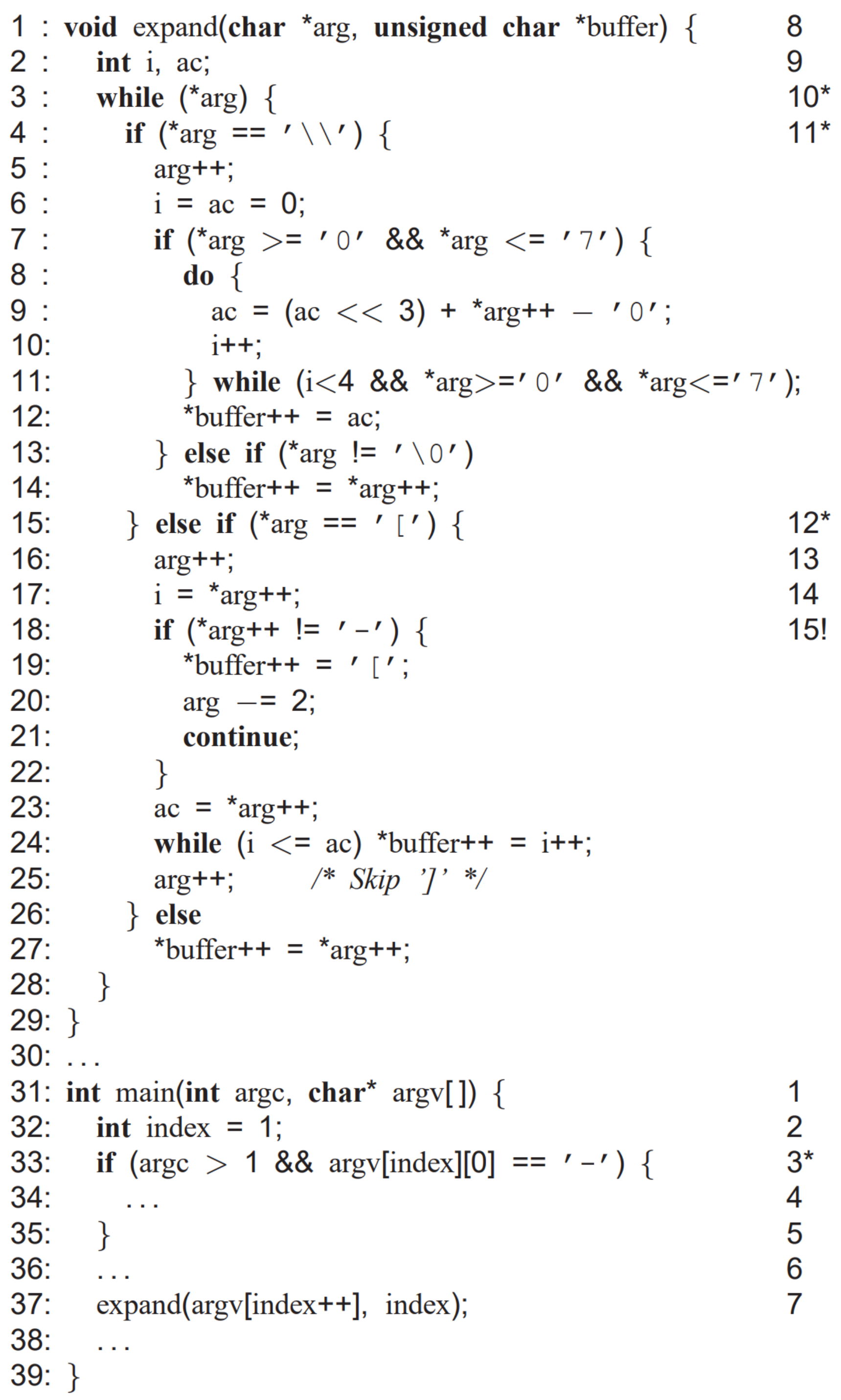
[cadar2008klee]

Impact (2): GNU Core Utils

89 binaries · 72k SLOC · 67.6% LCOV

```
klee <tool-name> \
    --max-time 60 \
    --sym-args 10 2 2 \
    --sym-files 2 8
```

[cadar2008klee]

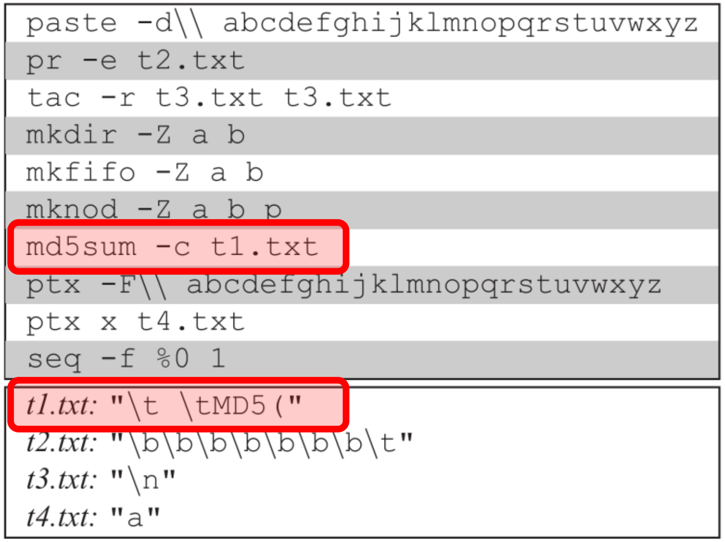


```
void expand(char *arg, unsigned char *buffer) {            8
  int i, ac;                                               9
  while (*arg) {                                           10*
    if (*arg == '\\') {                                    11*
      arg++;
      i = ac = 0;
      if (*arg >= '0' && *arg <= '7') {
        do {
          ac = (ac << 3) + *arg++ - '0';
          i++;
        } while (i<4 && *arg>='0' && *arg<='7');
        *buffer++ = ac;
      } else if (*arg != '\0')
        *buffer++ = *arg++;
    } else if (*arg == '[') {                              12*
      arg++;                                               13
      i = *arg++;                                          14
      if (*arg++ != '-') {                                 15!
        *buffer++ = '[';
        arg -= 2;
        continue;
      }
      ac = *arg++;
      while (i <= ac) *buffer++ = i++;
      arg++;       /* Skip ']' */
    } else
      *buffer++ = *arg++;
  }
}
...
int main(int argc, char* argv[]) {                         1
  int index = 1;                                           2
  if (argc > 1 && argv[index][0] == '-') {                 3*
    ...                                                    4
  }                                                        5
  ...                                                      6
  expand(argv[index++], index);                            7
  ...
}
```

```
tr [ "" ""
```

Impact (2): GNU Core Utils

89 binaries · 72k SLOC · 67.6% LCOV



```
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.

[cadar2008klee]

Impact (2): GNU Core Utils

- 89 binaries · 72k SLOC · 67.6% LCOV

- KLEE (2008):
  - +56 exploits / 89 h
  - 84.5% LCOV (in 1 h/app)
- Mayhem (2012):
  - 97.6% LVOC (in 1 h/app)

[cadar2008klee, cha2012unleashing]

Impact (3): Debian

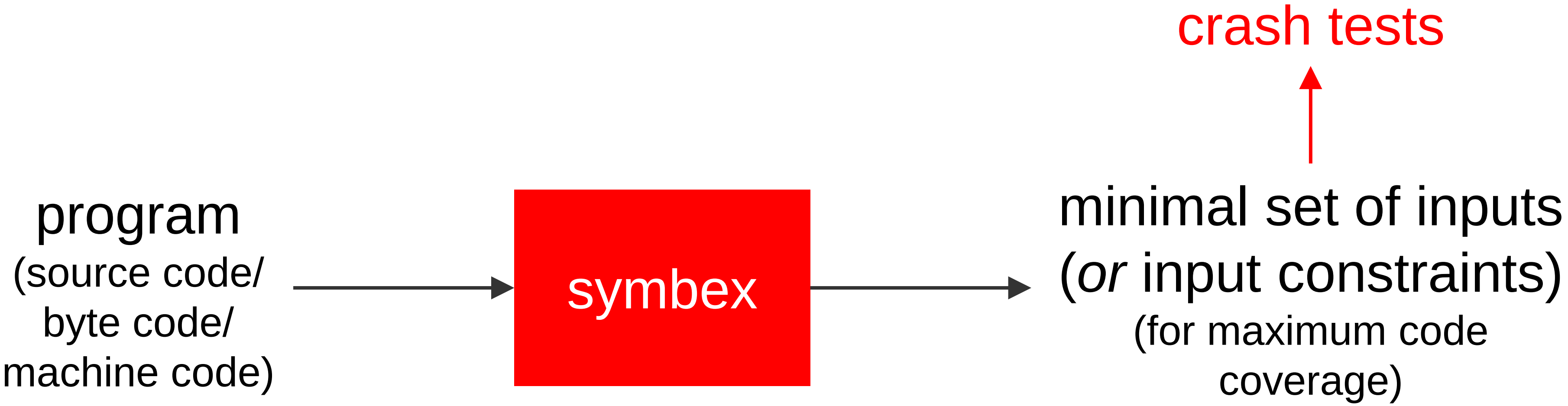
- 33k binaries · 679M SLOC
- MergePoint (2014): 11k exploits / 18 CPU-months
  - Amazon EC2: USD 0.28 / exploit
  - (zero-day market: up to USD 500,000.00 / exploit)

[avgerinos2014enhancing, debian]

Tools and Applications

Exploit Detection

- aka bug checking, vulnerability checking, exploit generation, (crash) test generation



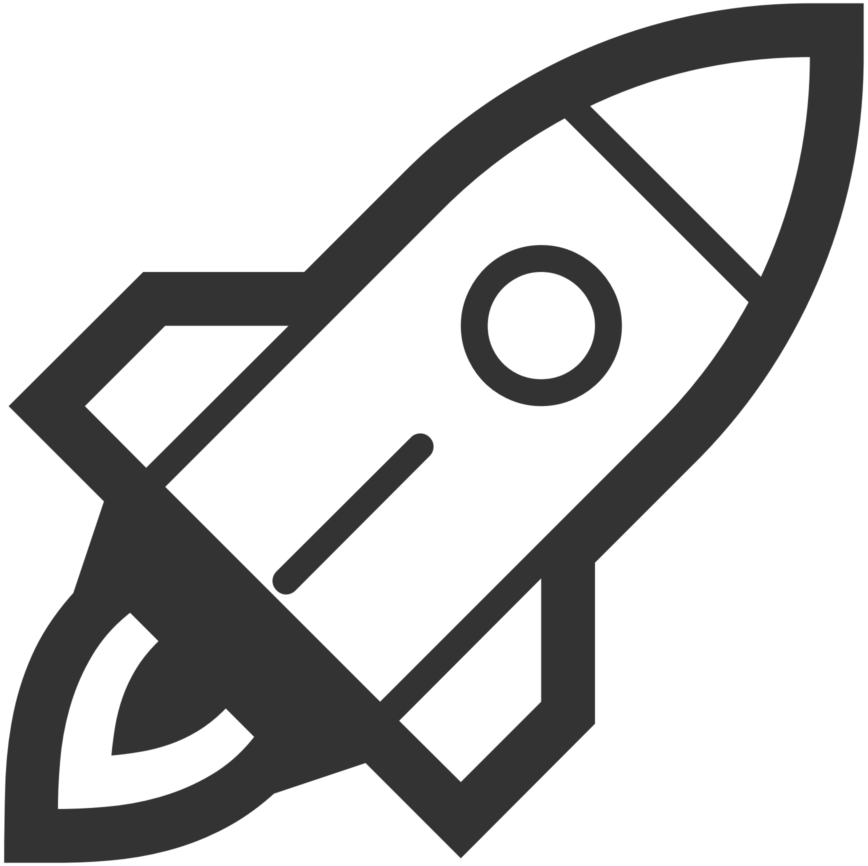
- crash test: just a serialized bug, no assertions

Further Use Cases For Program Analysis

- Assertion checking/invariant testing
  - Are there any read accesses to uninitialized memory?
  - How many SQL queries are executed per request to server?
  - Are there performed any elevated commands before the user has authorized?
  - …
- Dead code detection
- Invariant mining
  - anomaly detection
  - generation of unit tests/contracts
- Contract programming

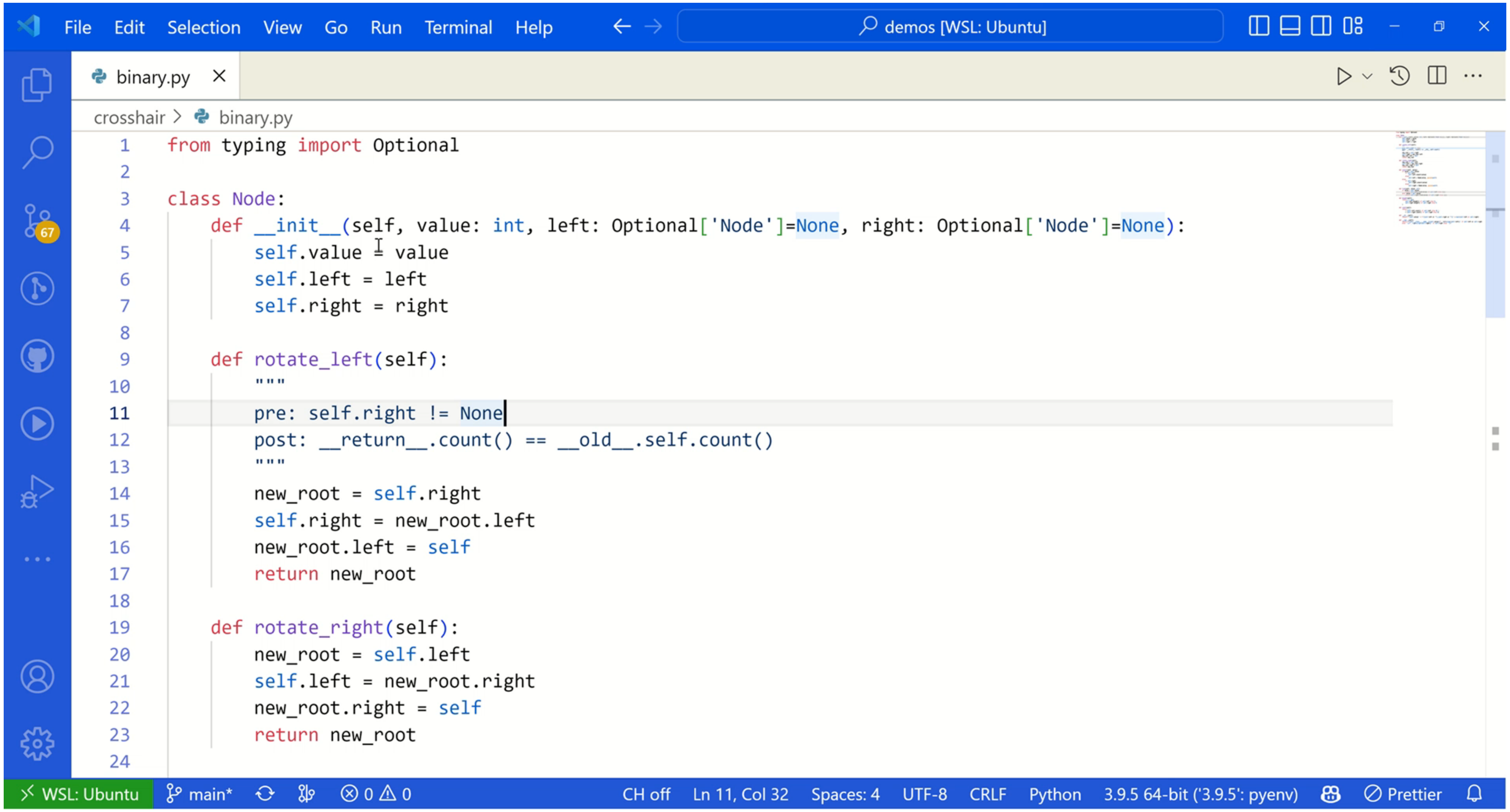
[csallner2008dysy, crosshair]

CrossHair



[crosshair]

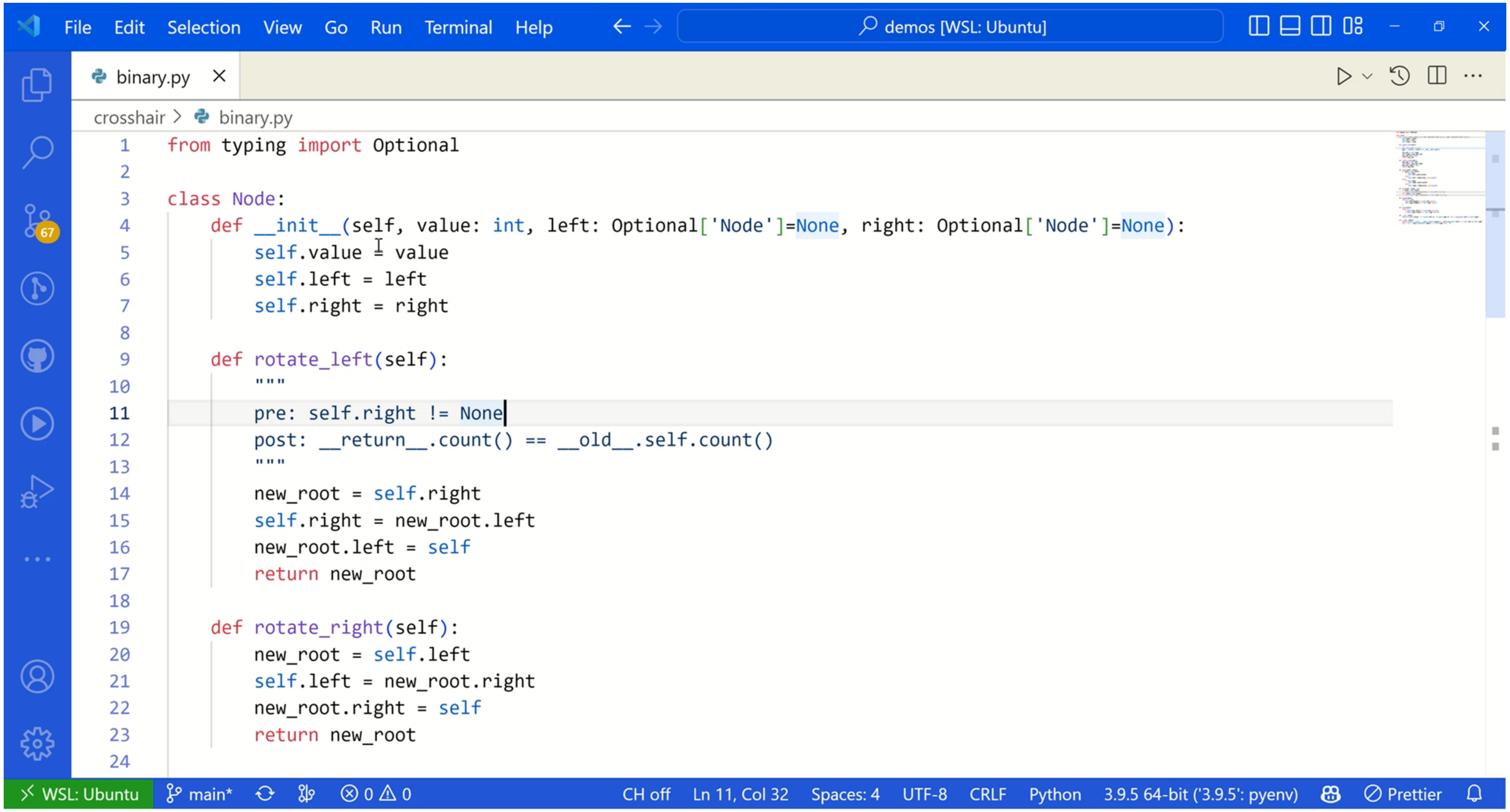
Demo: CrossHair (test generation)



```python
from typing import Optional

class Node:
    def __init__(self, value: int, left: Optional['Node']=None, right: Optional['Node']=None):
        self.value = value
        self.left = left
        self.right = right

    def rotate_left(self):
        """
        pre: self.right != None
        post: __return__.count() == __old__.self.count()
        """
        new_root = self.right
        self.right = new_root.left
        new_root.left = self
        return new_root

    def rotate_right(self):
        new_root = self.left
        self.left = new_root.right
        new_root.right = self
        return new_root

```

[crosshair]

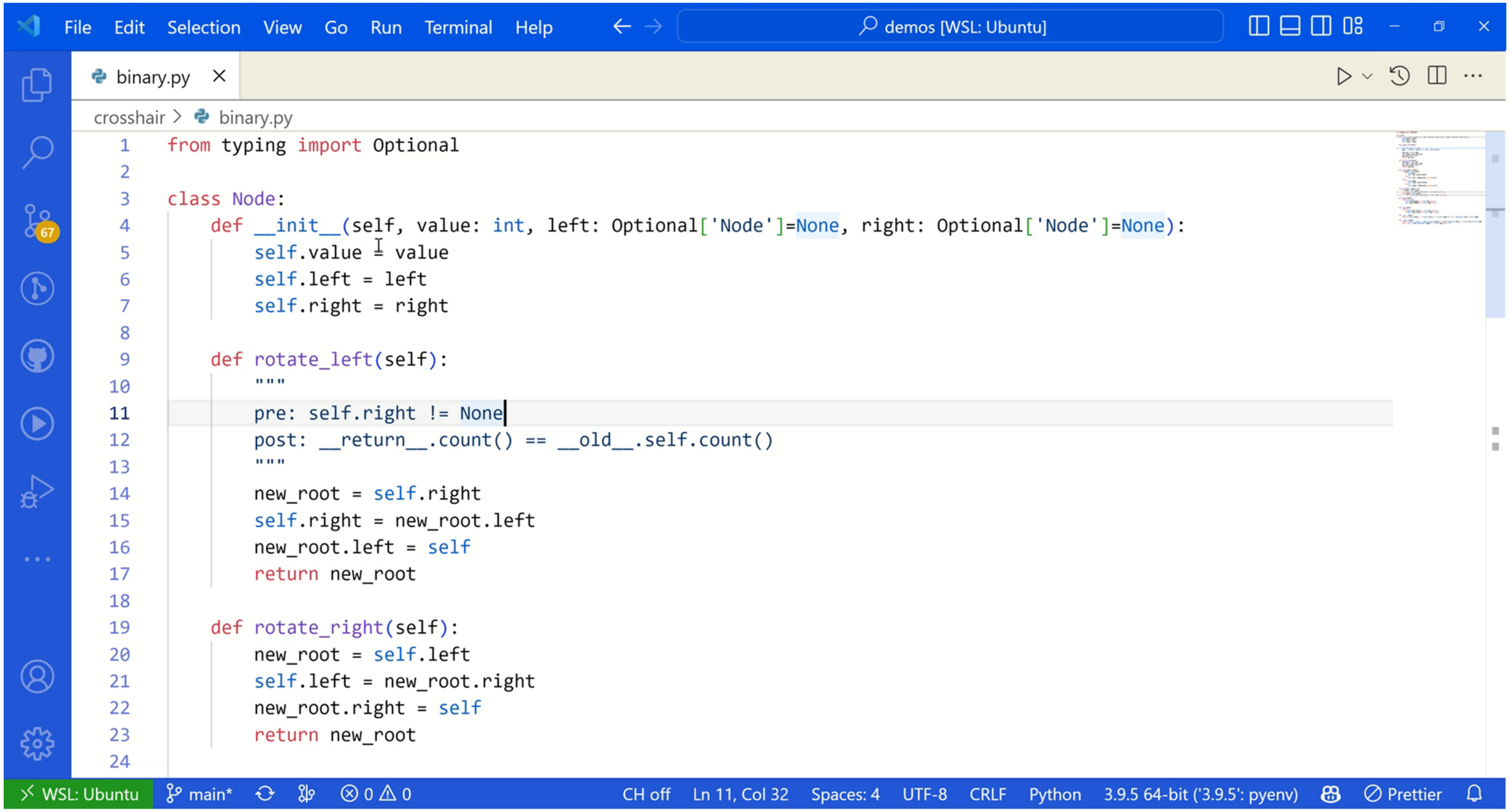
Demo: CrossHair (contract programming)



```
from typing import Optional

class Node:
    def __init__(self, value: int, left: Optional['Node']=None, right: Optional['Node']=None):
        self.value = value
        self.left = left
        self.right = right

    def rotate_left(self):
        """
        pre: self.right != None
        post: __return__.count() == __old__.self.count()
        """
        new_root = self.right
        self.right = new_root.left
        new_root.left = self
        return new_root

    def rotate_right(self):
        new_root = self.left
        self.left = new_root.right
        new_root.right = self
        return new_root

```

[crosshair]

Demo: CrossHair (behavioral diff)



```python
from typing import Optional

class Node:
    def __init__(self, value: int, left: Optional['Node']=None, right: Optional['Node']=None):
        self.value = value
        self.left = left
        self.right = right

    def rotate_left(self):
        """
        pre: self.right != None
        post: __return__.count() == __old__.self.count()
        """
        new_root = self.right
        self.right = new_root.left
        new_root.left = self
        return new_root

    def rotate_right(self):
        new_root = self.left
        self.left = new_root.right
        new_root.right = self
        return new_root

```

[crosshair]

CrossHair

- test generation
  - multiple coverage strategies
- contract programming with interactive assertions
- compare implementations

+

- aid for improving code coverage
- faster discovery of bugs
- other advantages of contract programming for larger teams and projects

–

- overhead for writing specification
  - preconditions/types and postconditions
- very limited practicability
  - insufficient performance
  - insufficient theories (e.g., for strings)

[crosshair]

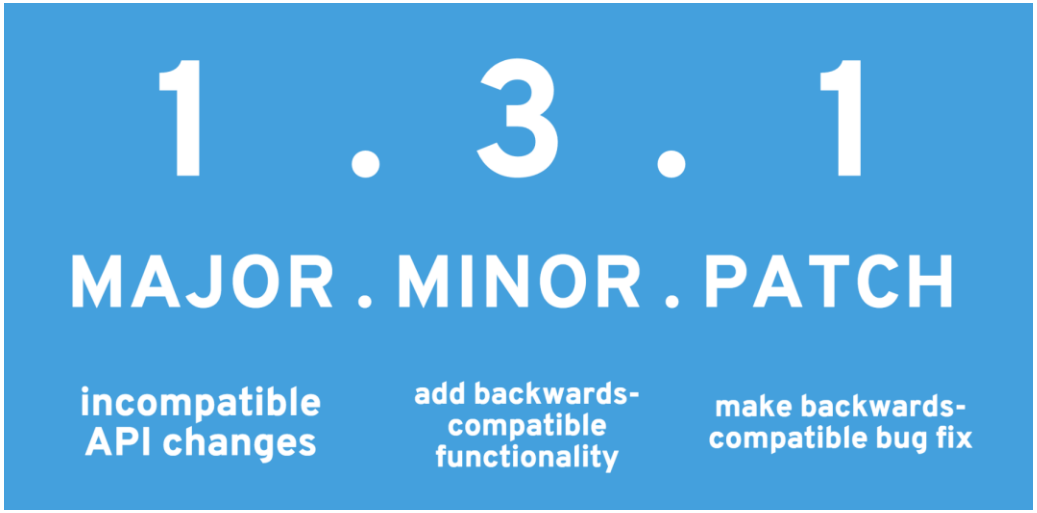
Compare Implementations

- assert foo(*args) == bar(*args)

[crosshair]

Contractual SemVer Verification

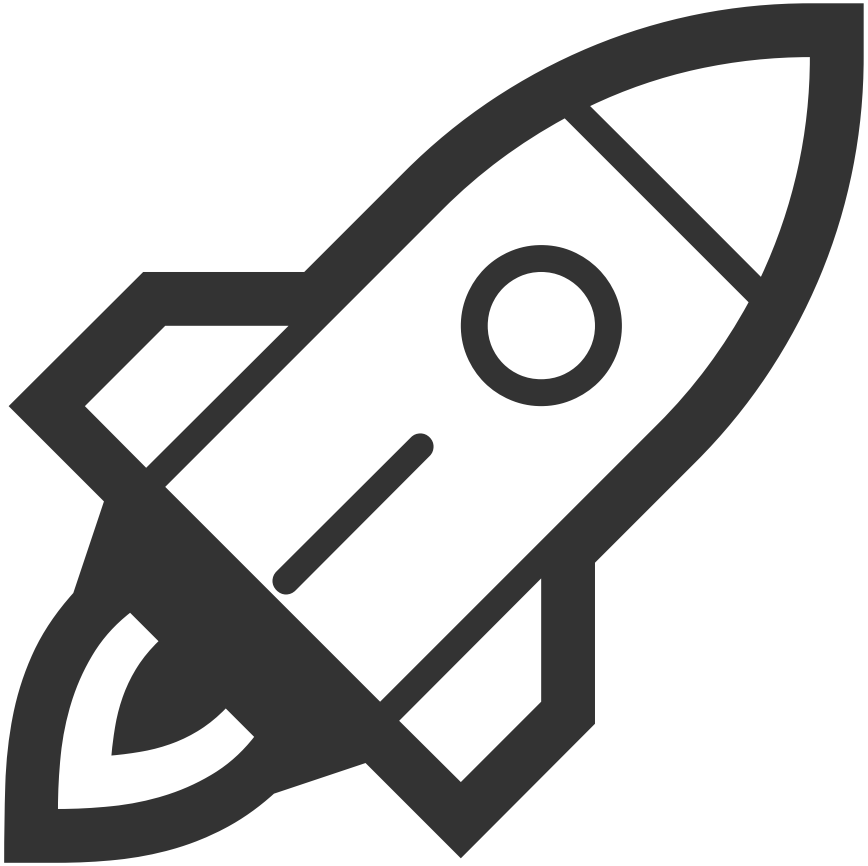
- SemVer (Semantic Versioning): versioning scheme for backward compatibility


1 . 3 . 1
MAJOR . MINOR . PATCH
incompatible API changes
add backwards-compatible functionality
make backwards-compatible bug fix

- Contractual SemVer: SemVer with compatibility definitions based on formal contracts
- Verification: symbolic comparison of versions

[crosshair, schanely2022contractual]

Microsoft IntelliTest



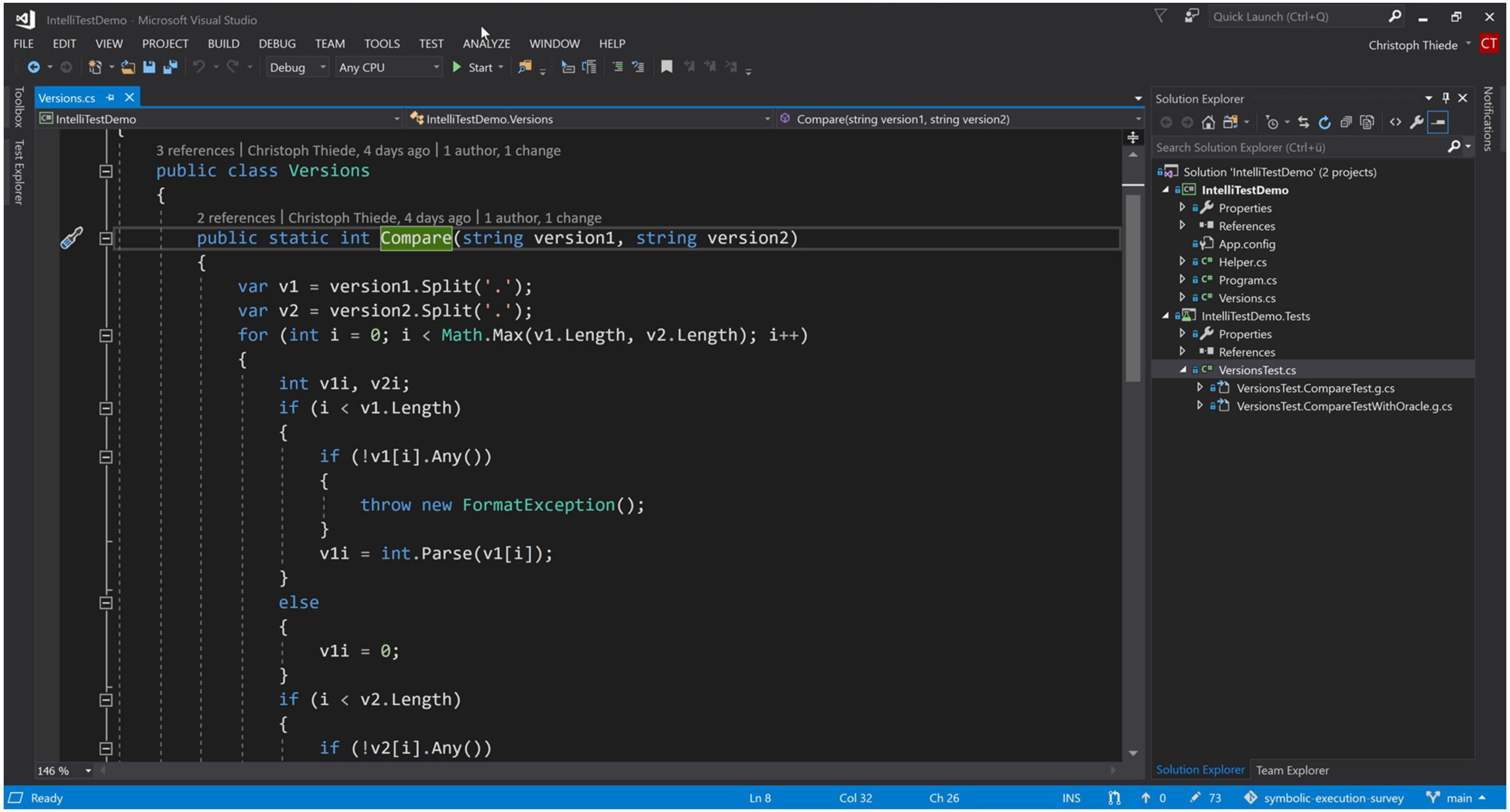
[intellitest, tillmann2008pex]

Demo: Microsoft IntelliTest

- explore method: display input/output table (compare versions)
- generate parametrized unit test (compare versions)
  - assumptions and assertions
  - show generated source code
  - automated type choice (IComparer)
  - automatic mocking (PexPConsoleInContext)
- more examples -> performance limits
  - regex
  - hashes
  - factorization
- further observations
  - better performance, but still limited
  - interactive exploration/reverse engineering
  - many conveniences, still hard to deal with context
    - overhead for specifying mocks/factories
    - only simply applicable for methods with little context
  - too many contingencies: Console.WriteLine() might throw several exceptions one might not want to handle
    - configuration overhead

[intellitest, tillmann2008pex]

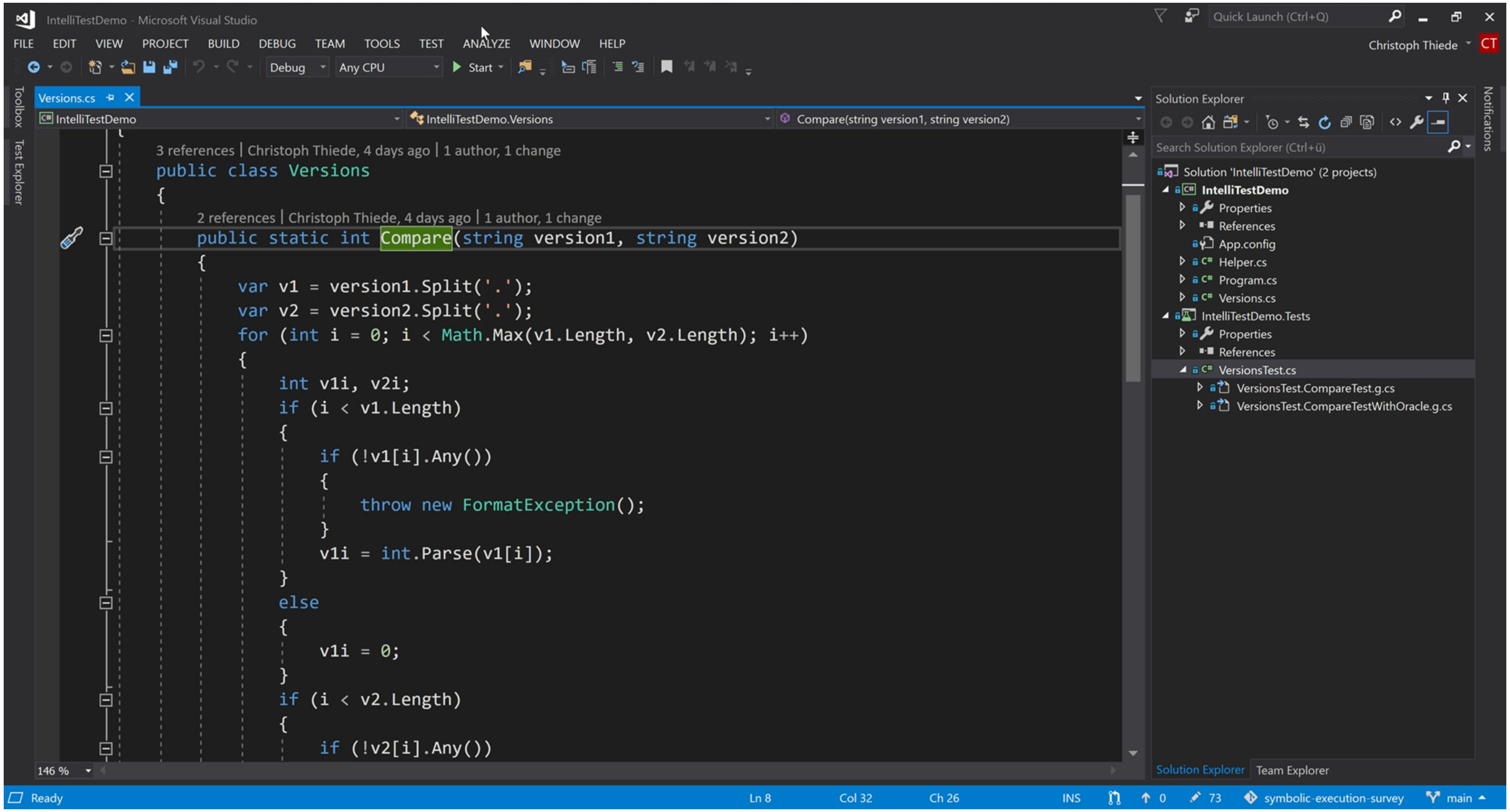
Demo: Microsoft IntelliTest (exploration)



```
3 references | Christoph Thiede, 4 days ago | 1 author, 1 change
public class Versions
{
    2 references | Christoph Thiede, 4 days ago | 1 author, 1 change
    public static int Compare(string version1, string version2)
    {
        var v1 = version1.Split('.');
        var v2 = version2.Split('.');
        for (int i = 0; i < Math.Max(v1.Length, v2.Length); i++)
        {
            int v1i, v2i;
            if (i < v1.Length)
            {
                if (!v1[i].Any())
                {
                    throw new FormatException();
                }
                v1i = int.Parse(v1[i]);
            }
            else
            {
                v1i = 0;
            }
            if (i < v2.Length)
            {
                if (!v2[i].Any())
```

[intellitest, tillmann2008pex]

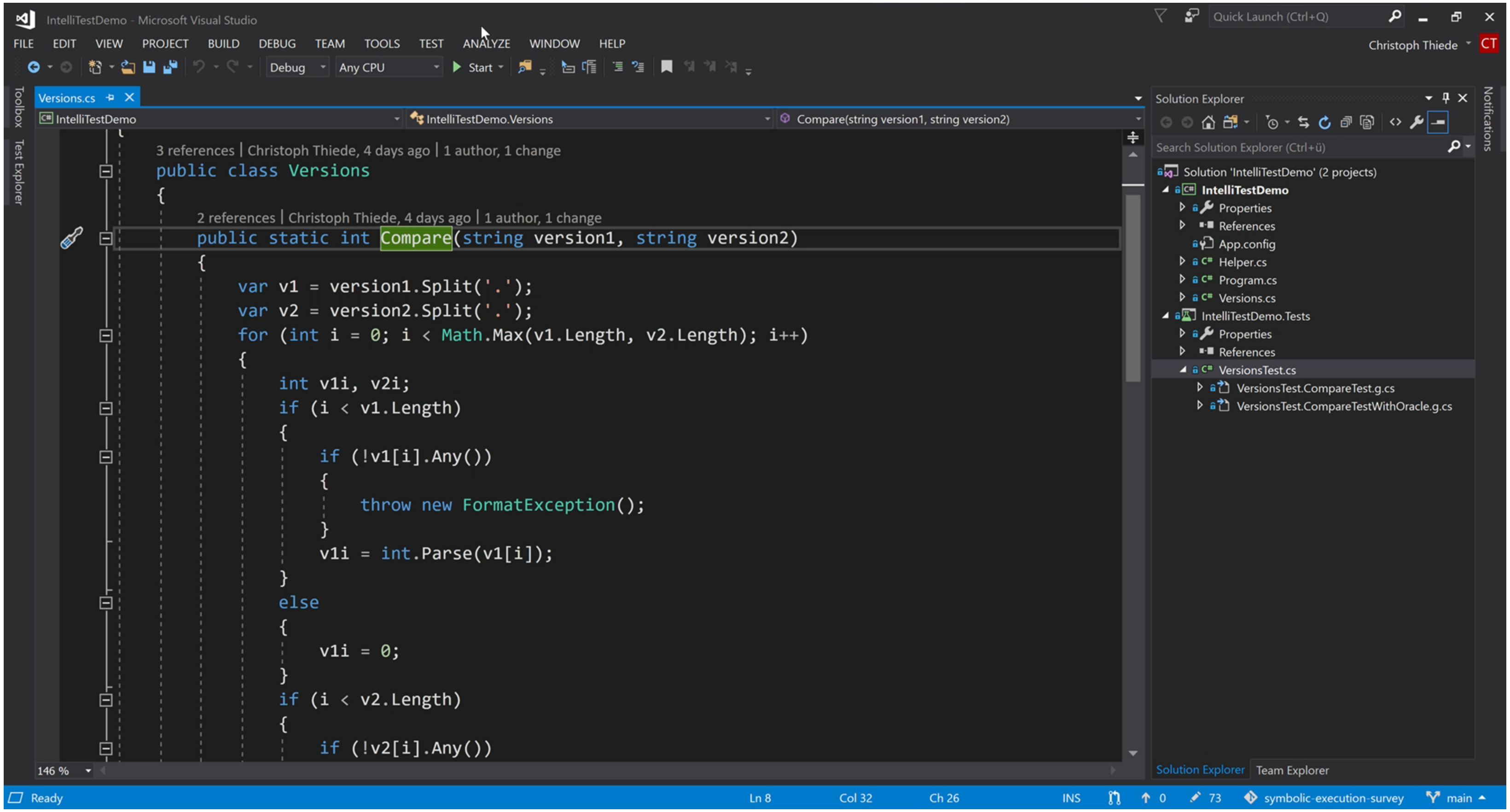
Demo: Microsoft IntelliTest



```
3 references | Christoph Thiede, 4 days ago | 1 author, 1 change
public class Versions
{
    2 references | Christoph Thiede, 4 days ago | 1 author, 1 change
    public static int Compare(string version1, string version2)
    {
        var v1 = version1.Split('.');
        var v2 = version2.Split('.');
        for (int i = 0; i < Math.Max(v1.Length, v2.Length); i++)
        {
            int v1i, v2i;
            if (i < v1.Length)
            {
                if (!v1[i].Any())
                {
                    throw new FormatException();
                }
                v1i = int.Parse(v1[i]);
            }
            else
            {
                v1i = 0;
            }
            if (i < v2.Length)
            {
                if (!v2[i].Any())
```

[intellitest, tillmann2008pex]

Demo: Microsoft IntelliTest



```
3 references | Christoph Thiede, 4 days ago | 1 author, 1 change
public class Versions
{
    2 references | Christoph Thiede, 4 days ago | 1 author, 1 change
    public static int Compare(string version1, string version2)
    {
        var v1 = version1.Split('.');
        var v2 = version2.Split('.');
        for (int i = 0; i < Math.Max(v1.Length, v2.Length); i++)
        {
            int v1i, v2i;
            if (i < v1.Length)
            {
                if (!v1[i].Any())
                {
                    throw new FormatException();
                }
                v1i = int.Parse(v1[i]);
            }
            else
            {
                v1i = 0;
            }
            if (i < v2.Length)
            {
                if (!v2[i].Any())
```

[intellitest, tillmann2008pex]

Microsoft IntelliTest

- parametrized unit testing (PUT)
- input/output table for exploration
- framework and predefined suite of mocks and stubs

+

- aid for improving code coverage
- exploration of program behavior and edge cases
- better performance and practicability

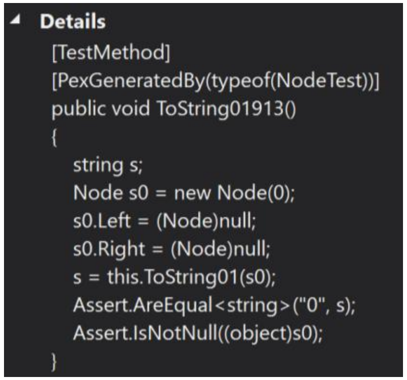
–

- still limited performance

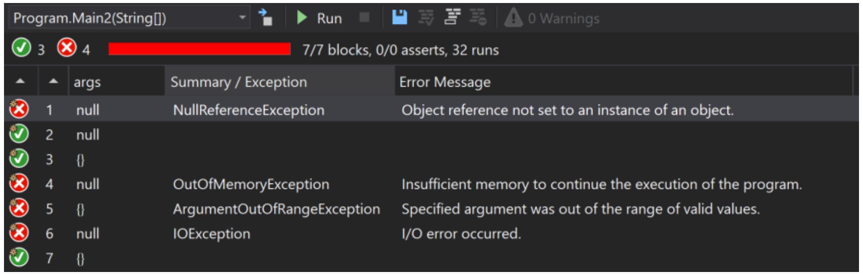
[intellitest, tillmann2008pex]

Test Generation

- Limitations:
  - tests suite size vs code quality
    - readability/documentation of fixtures
    - code reuse/idiomacity of setup code
    - robustness against future refactorings
  - choice of concrete values
    - should provide intuition
    - -1425360904 vs -1
    - '厧ძ▪䀕' vs ‘abcd’ vs ‘John’
  - too many contingencies
    - uncommon exceptions
    - implicit contracts
    - configuration overhead
  - missing context
    - overhead for specifying mocks/factories
    - low entry barrier only for any method/unit with little context



```
◢ Details
    [TestMethod]
    [PexGeneratedBy(typeof(NodeTest))]
    public void ToString01913()
    {
        string s;
        Node s0 = new Node(0);
        s0.Left = (Node)null;
        s0.Right = (Node)null;
        s = this.ToString01(s0);
        Assert.AreEqual<string>("0", s);
        Assert.IsNotNull((object)s0);
    }
```

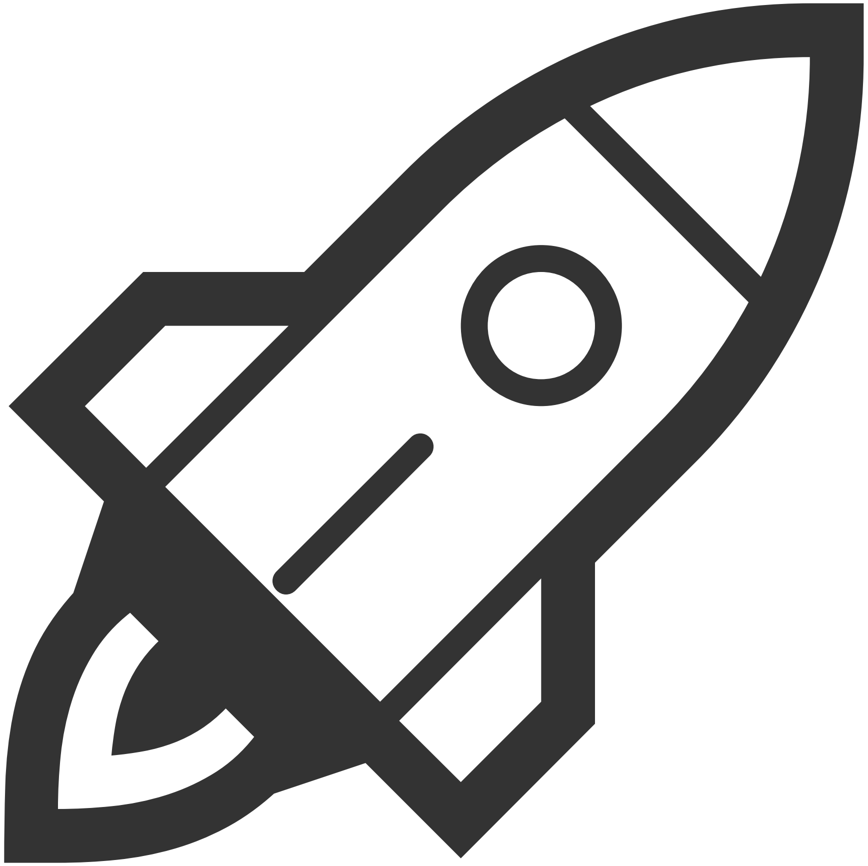


✓ 3 ✗ 4 7/7 blocks, 0/0 asserts, 32 runs

		args	Summary / Exception	Error Message
✗	1	null	NullReferenceException	Object reference not set to an instance of an object.
✓	2	null		
✓	3	{}		
✗	4	null	OutOfMemoryException	Insufficient memory to continue the execution of the program.
✗	5	{}	ArgumentOutOfRangeException	Specified argument was out of the range of valid values.
✗	6	null	IOException	I/O error occurred.
✓	7	{}		

[crosshair, intellitest, tillmann2008pex]

Symbolic Execution Debugger (SED)



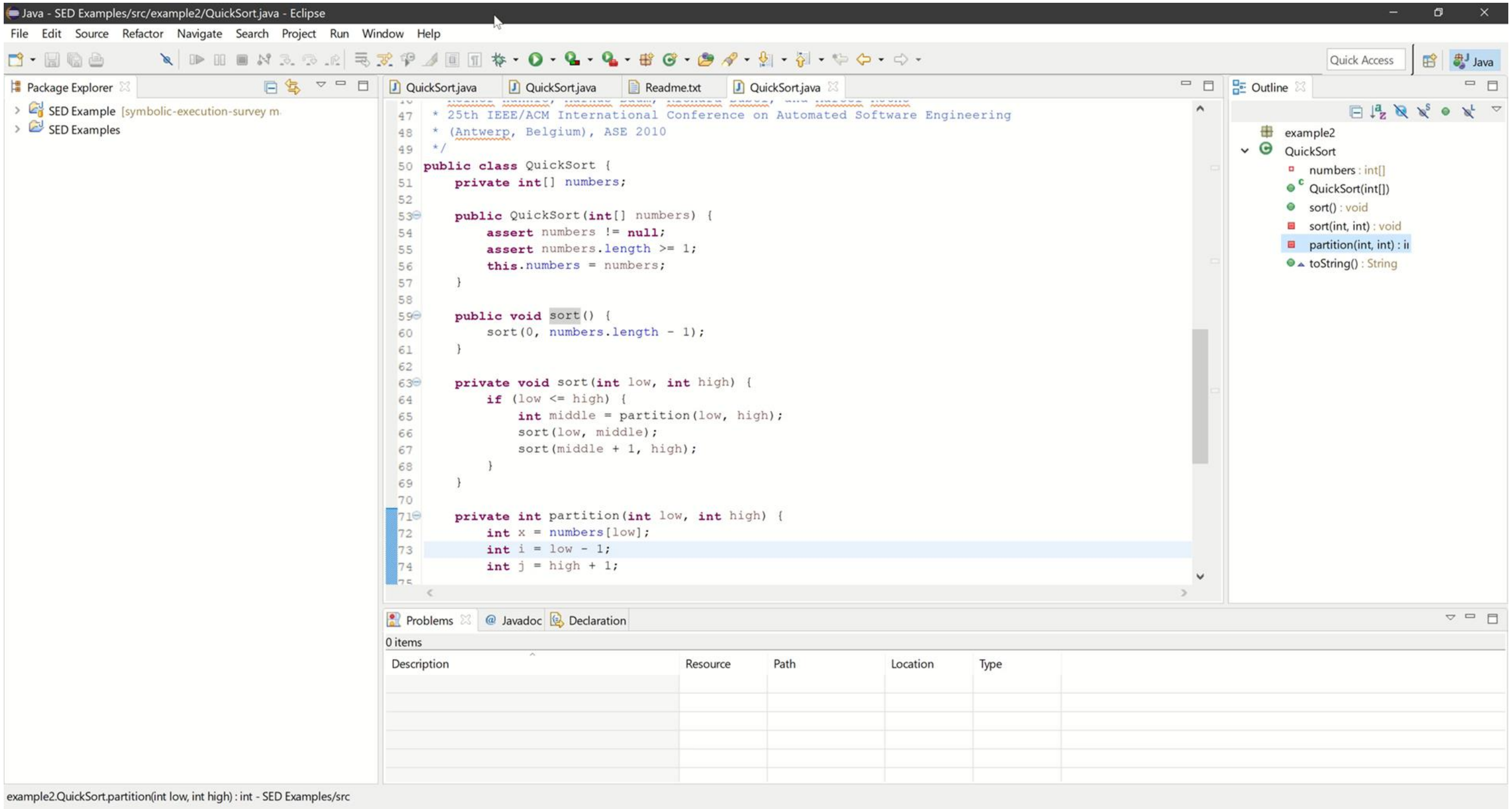
[hentschel2019symbolic, sed]

Demo: Symbolic Execution Debugger (SED)

- debug QuickSort
  - navigate through and expand execution tree
  - see symbolic variables
- observations:
  - no entry point required
  - overcrowded, too many branches
  - still need to specify much missing context (sometimes optional only to clean up execution tree)
  - hard to follow control statements (e.g., loops)
  - potentially helpful to evade too much irrelevant context

[hentschel2019symbolic, sed]

Demo: Symbolic Execution Debugger (SED)



```java
 * 25th IEEE/ACM International Conference on Automated Software Engineering
 * (Antwerp, Belgium), ASE 2010
 */
public class QuickSort {
    private int[] numbers;

    public QuickSort(int[] numbers) {
        assert numbers != null;
        assert numbers.length >= 1;
        this.numbers = numbers;
    }

    public void sort() {
        sort(0, numbers.length - 1);
    }

    private void sort(int low, int high) {
        if (low <= high) {
            int middle = partition(low, high);
            sort(low, middle);
            sort(middle + 1, high);
        }
    }

    private int partition(int low, int high) {
        int x = numbers[low];
        int i = low - 1;
        int j = high + 1;
```

example2.QuickSort.partition(int low, int high) : int - SED Examples/src

[hentschel2019symbolic, sed]

Symbolic Execution Debugger (SED)

- interactive exploration and advancement of execution tree
- inspect symbolic variables
- visualize memory layouts

+

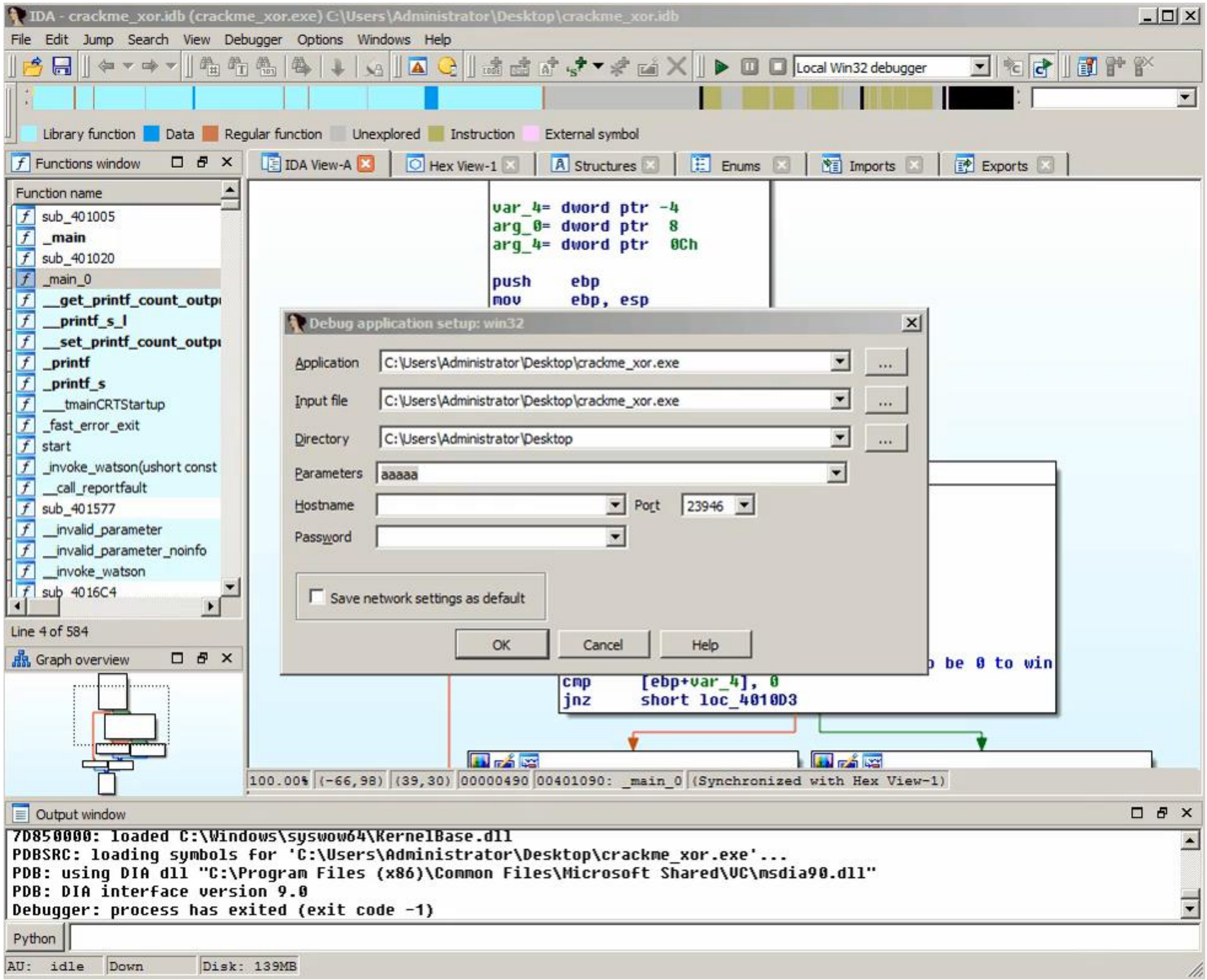
- debugging without entrypoint
- evade irrelevant / distracting context

–

- overcrowded tree
- still often need to specify missing context
  - sometimes optional only to clean up execution tree
- hard to follow control statements in tree
  - e.g., loops

[hentschel2019symbolic, sed]

Symbolic Execution Debugging: Ponce



IDA - crackme_xor.idb (crackme_xor.exe) C:\Users\Administrator\Desktop\crackme_xor.idb

- sub_401005
- _main
- sub_401020
- _main_0
- __get_printf_count_outp
- __printf_s_l
- __set_printf_count_outp
- _printf
- _printf_s
- __tmainCRTStartup
- _fast_error_exit
- start
- _invoke_watson(ushort const
- __call_reportfault
- sub_401577
- __invalid_parameter
- __invalid_parameter_noinfo
- __invoke_watson
- sub_4016C4

Line 4 of 584

```
var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
```

Debug application setup: win32

Application	C:\Users\Administrator\Desktop\crackme_xor.exe
Input file	C:\Users\Administrator\Desktop\crackme_xor.exe
Directory	C:\Users\Administrator\Desktop
Parameters	aaaaa
Hostname	
Port	23946
Password	

□ Save network settings as default

```
be 0 to win
cmp     [ebp+var_4], 0
jnz     short loc_4010D3
```

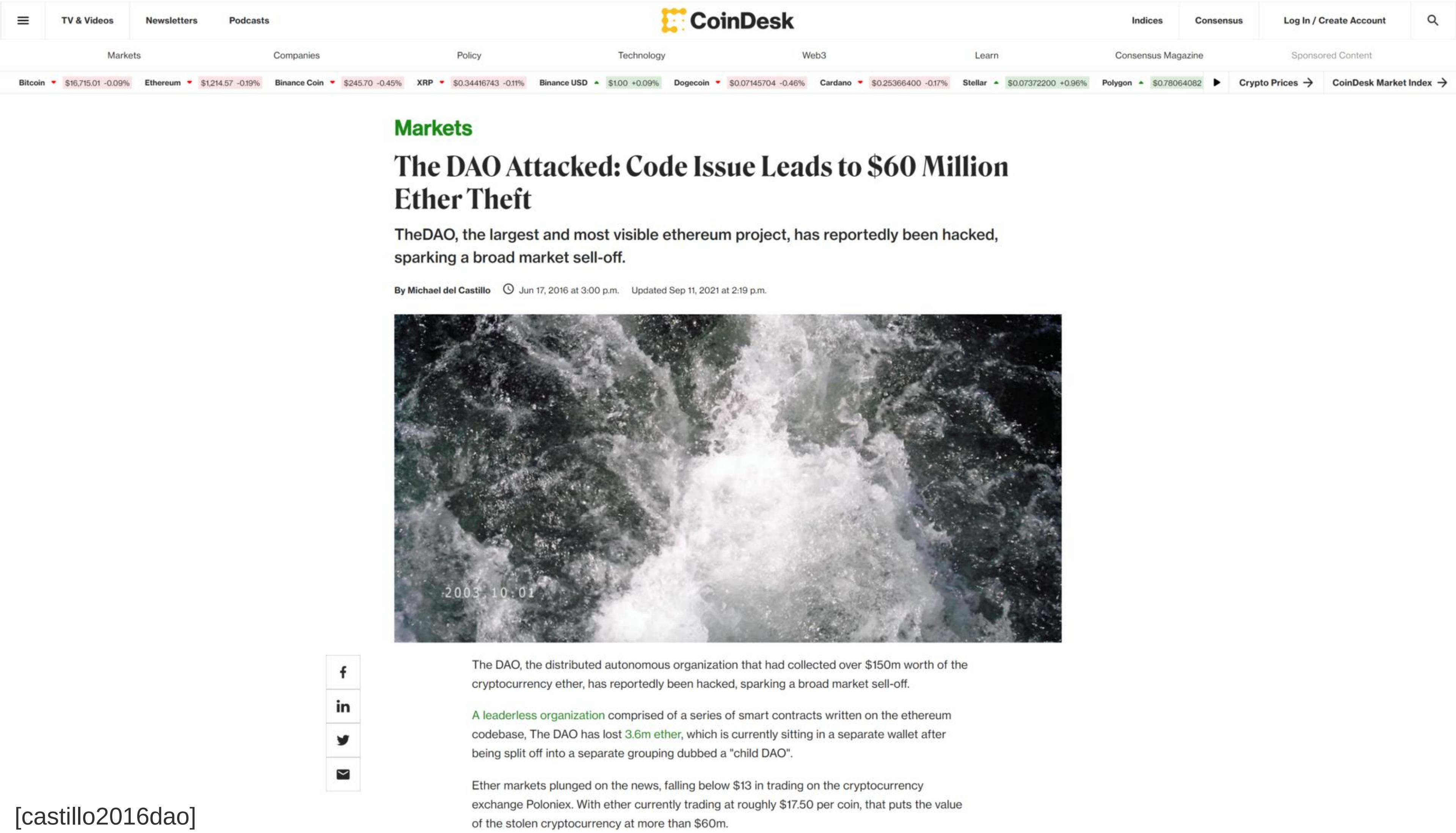
```
100.00% (-66,98) (39,30) 00000490 00401090: _main_0 (Synchronized with Hex View-1)
```

```
7D850000: loaded C:\Windows\syswow64\KernelBase.dll
PDBSRC: loading symbols for 'C:\Users\Administrator\Desktop\crackme_xor.exe'...
PDB: using DIA dll "C:\Program Files (x86)\Common Files\Microsoft Shared\VC\msdia90.dll"
PDB: DIA interface version 9.0
Debugger: process has exited (exit code -1)
```

```
AU:  idle   Down   Disk: 139MB
```

[ponce]

Smart Contract Validation



Markets

The DAO Attacked: Code Issue Leads to $60 Million Ether Theft

TheDAO, the largest and most visible ethereum project, has reportedly been hacked, sparking a broad market sell-off.

By Michael del Castillo Jun 17, 2016 at 3:00 p.m. Updated Sep 11, 2021 at 2:19 p.m.

The DAO, the distributed autonomous organization that had collected over $150m worth of the cryptocurrency ether, has reportedly been hacked, sparking a broad market sell-off.

A leaderless organization comprised of a series of smart contracts written on the ethereum codebase, The DAO has lost 3.6m ether, which is currently sitting in a separate wallet after being split off into a separate grouping dubbed a "child DAO".

Ether markets plunged on the news, falling below $13 in trading on the cryptocurrency exchange Poloniex. With ether currently trading at roughly $17.50 per coin, that puts the value of the stolen cryptocurrency at more than $60m.

[castillo2016dao]

Dynamic Recompilation

- context: automatic patching of binaries without source code
  - security patches
  - optimizations
- deny changes in behavior through symbolic execution

[altinay2020binrec]

Tooling Impact

- Dynamic analysis frameworks
  - angr, Manticore, Miasm, Triton, … (>2k stars on GitHub)
- Disassembly
  - Ponce, Medusa (>1k stars on GitHub)
- Testing and exploration
  - Microsoft IntelliTest (available to millions of Visual Studio users)
  - CrossHair, DeepState (>700 stars on GitHub)

Suitability

- So when does it make sense to use symbolic execution for my project?

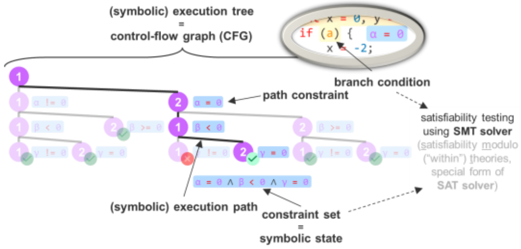
- For test generation: consider
  - complexity of data and computations
  - degree of coupling to environment (frameworks, APIs, I/O, …)
  - your computational and temporal resources
  - your quality requirements and security policy
- For interactive programming feedback:
  - usual trade-offs for linter-like tools (e.g., immediacy of feedback vs distractions)
- For reverse engineering/disassembling tasks:
  - Symbex debugging/exploration tools can support the understanding of poor-readable code bases.

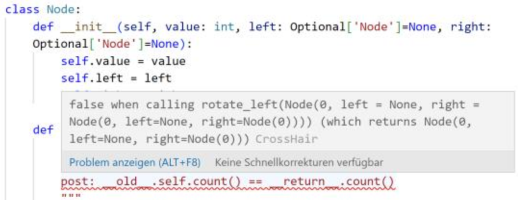
Alternatives

- Static analysis tools:
  - better performance
  - lower precision/selectivity
  - often more and more mature solutions available
- Fuzzing:
  - better performance
  - lower precision/selectivity
- Generative AI tools:
  - e.g., GitHub Copilot, ChatGPT for finding/fixing bugs or generating tests
  - depending on popularity of domain, possibly higher efficiency than manual approaches
  - still, quality of results is unreliable and overconfident, awkward workflow

Conclusion

- Symbolic execution: dynamic program analysis technique to check (almost) all program paths
- Tools for bug detection, unit testing, and program exploration exist in many programming languages
- Used successfully for various popular software
- Systemic limitations (path explosion, unsolvable constraint sets, symbolic pointers, …)
- Overhead for specifying tests/mocks

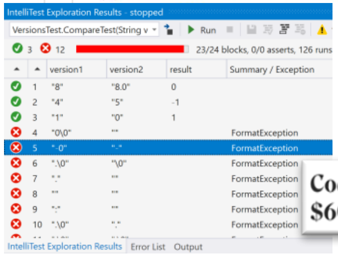




```
class Node:
    def __init__(self, value: int, left: Optional['Node']=None, right:
    Optional['Node']=None):
        self.value = value
        self.left = left
```

```
false when calling rotate_left(Node(0, left = None, right =
Node(0, left=None, right=Node(0)))) (which returns Node(0,
left=None, right=Node(0))) CrossHair
```

```
    def
        post: __old__.self.count() == __return__.count()
        """
```


IntelliTest Exploration Results - stopped
VersionsTest.CompareTest(String v
Run

✓ 3 ✗ 12 23/24 blocks, 0/0 asserts, 126 runs

		version1	version2	result	Summary / Exception
✓	1	"8"	"8.0"	0	
✓	2	"4"	"5"	-1	
✓	3	"1"	"0"	1	
✗	4	"0\0"	""		FormatException
✗	5	"-0"	"-"		FormatException
✗	6	".\0"	"\0"		FormatException
✗	7	"."	""		FormatException
✗	8	""	""		FormatException
✗	9	":"	""		FormatException
✗	10	".\0"	"."		FormatException

IntelliTest Exploration Results
Error List
Output

Code Issue Leads to $60 Million Ether Theft

```
class VirtualFileSystem {…}
```

Reading

LinqLover/symbolic-execution-survey

Additional notes

Documented examples

Bibliography

coming soon

Report

Literature: Publications

[altinay2020binrec] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, and others. 2020. BinRec: dynamic binary lifting and recompilation. In Proceedings of the Fifteenth European Conference on Computer Systems, 1–16.

[avgerinos2014enhancing] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In Proceedings of the 36th International Conference on Software Engineering, 1083–1094.

[baldoni2018survey] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) 51, 3 (2018), 1–39.

[cadar2005execution] Cristian Cadar and Dawson Engler. 2005. Execution generated test cases: How to make systems code crash itself. In International SPIN Workshop on Model Checking of Software, Springer, 2–23.

[cadar2008klee] Cristian Cadar, Daniel Dunbar, Dawson R Engler, and others. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, 209–224.

[cadar2013symbolic] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. Communications of the ACM 56, 2 (2013), 82–90.

[cha2012unleashing] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In 2012 IEEE Symposium on Security and Privacy, IEEE, 380–394.

[csallner2008dysy] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy. In 2008 ACM/IEEE 30th International Conference on Software Engineering, IEEE, 281–290.

[godefroid2008automated] Patrice Godefroid, Michael Y Levin, David A Molnar, and others. 2008. Automated whitebox fuzz testing. In NDSS, 151–166.

[godefroid2012sage] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft. Queue 10, 1 (2012), 20–27.

[hentschel2019symbolic] Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2019. The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. International Journal on Software Tools for Technology Transfer 21, 5 (2019), 485–513.

[kuznetsov2012efficient] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. Acm Sigplan Notices 47, 6 (2012), 193–204.

[liu2017survey] Yu Liu, Xu Zhou, and Wei-Wei Gong. 2017. A survey of search strategies in the dynamic symbolic execution. In ITM Web of Conferences, EDP Sciences, 03025.

[mirzaei2012testing] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing android apps through symbolic execution. ACM SIGSOFT Software Engineering Notes 37, 6 (2012), 1–5.

[schanely2022contractual] Phillip Schanely. 2022. Contractual SemVer. Retrieved January 23, 2023 from https://github.com/pschanely/contractual-semver

[tillmann2008pex] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. In Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings 2, Springer, 134–153.

[yang2019advances] Guowei Yang, Antonio Filieri, Mateus Borges, Donato Clun, and Junye Wen. 2019. Advances in symbolic execution. Advances in Computers 113, (2019), 225–287.

Preliminary bibliography. Visit https://github.com/LinqLover/symbolic-execution-survey for the latest version of the bibliography.

Literature: Weblinks

[crosshair] Phillip Schanely and others. 2019 – 2022. CrossHair Documentation. Retrieved January 23, 2023 from https://crosshair.readthedocs.io/en/latest/

[debian] Statistics – Debian Sources. Retrieved January 23, 2023 from https://sources.debian.org/stats/

[dimensions] Timeline - Overview for “symbolic execution” in Publications – Dimensions. Retrieved January 23, 2023 from https://app.dimensions.ai/analytics/publication/overview/timeline?search_mode=content&search_text=%22symbolic%20execution%22&search_type=kws&search_field=full_search&year_from=1974&year_to=2023

[intellitest] G. Hogenson, G. Warren, T. Sherer, and others. 2017–2022. Overview of Microsoft IntelliTest. Retrieved January 23, 2023 from https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual

[luckow2017awesome] Kasper Luckow et al. 2017 – 2022. Awesome Symbolic Execution. Retrieved January 23, 2023 from https://github.com/ksluckow/awesome-symbolic-execution

[ponce] Alberto Garcia Illera and Francisco Oca. 2016-2022. Ponce. Retrieved January 23, 2023 from https://github.com/illera88/Ponce

[sed] Symbolic Execution Debugger (SED). Retrieved January 23, 2023 from https://www.key-project.org/eclipse/sed

[castillo2016dao] Michael del Castillo. 2016. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. Retrieved January 23, 2023 from https://www.coindesk.com/markets/2016/06/17/the-dao-attacked-code-issue-leads-to-60-million-ether-theft/

Preliminary bibliography. Visit https://github.com/LinqLover/symbolic-execution-survey for the latest version of the bibliography.