

SYMBOLIC EXECUTION

PROF. BRENDAN SALTAFORMAGGIO

SCHOOL OF ECE

CREATING THE NEXT®



PLEASE CONSIDER THE
ENVIRONMENT, AVOID
PRINTING SLIDES!

MANY THANKS TO ZHIQIANG LIN
AND MATHIAS PAYER FOR THEIR
CONTRIBUTIONS TO THESE SLIDES

WHAT IS SYMBOLIC EXECUTION?



- Symbolic execution (also symbolic analysis) is a means of analyzing a program to determine what inputs cause each part of a program to execute
 - Similar to slicing, but instead of concrete values we will use constraints
- Symbolic analysis steps through a program (for us: a binary) and performs abstract interpretation of the statements (for us: instructions)
 - Data values are turned into formulas based on how they are used in the program
 - Each execution path is represented via constraints (sets of formulas on the accessed data)
- “An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would, a case of abstract interpretation. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.”
 - https://en.wikipedia.org/wiki/Symbolic_execution

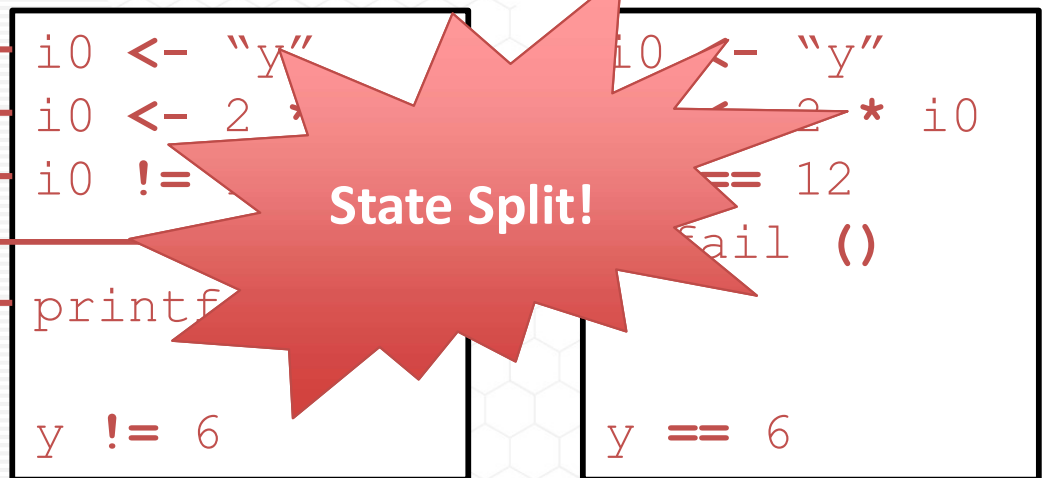
BUT... WHY?

- James C. King. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394.
- Perform analysis of programs WITHOUT inputs
 - Execute a program on symbolic inputs
 - Build constraints on the input values that drive the execution to each path
 - For example, constraints could say "Variable x must be 5 to execute this path"
 - Key Insight: Code can generate its own test cases
- Security Applications:
 - Malware analysis without input and get full code coverage (maybe)!
 - Vulnerability finding --- constraint says "Input 'ABC' will cause buffer to overflow"
 - Exploit generation --- constraint says "Input 'ABC' will cause RIP to jump"
- Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. "Automatic exploit generation." Communications of the ACM 57.2 (2014): 74-84.

SYMBOLIC EXECUTION EXAMPLE

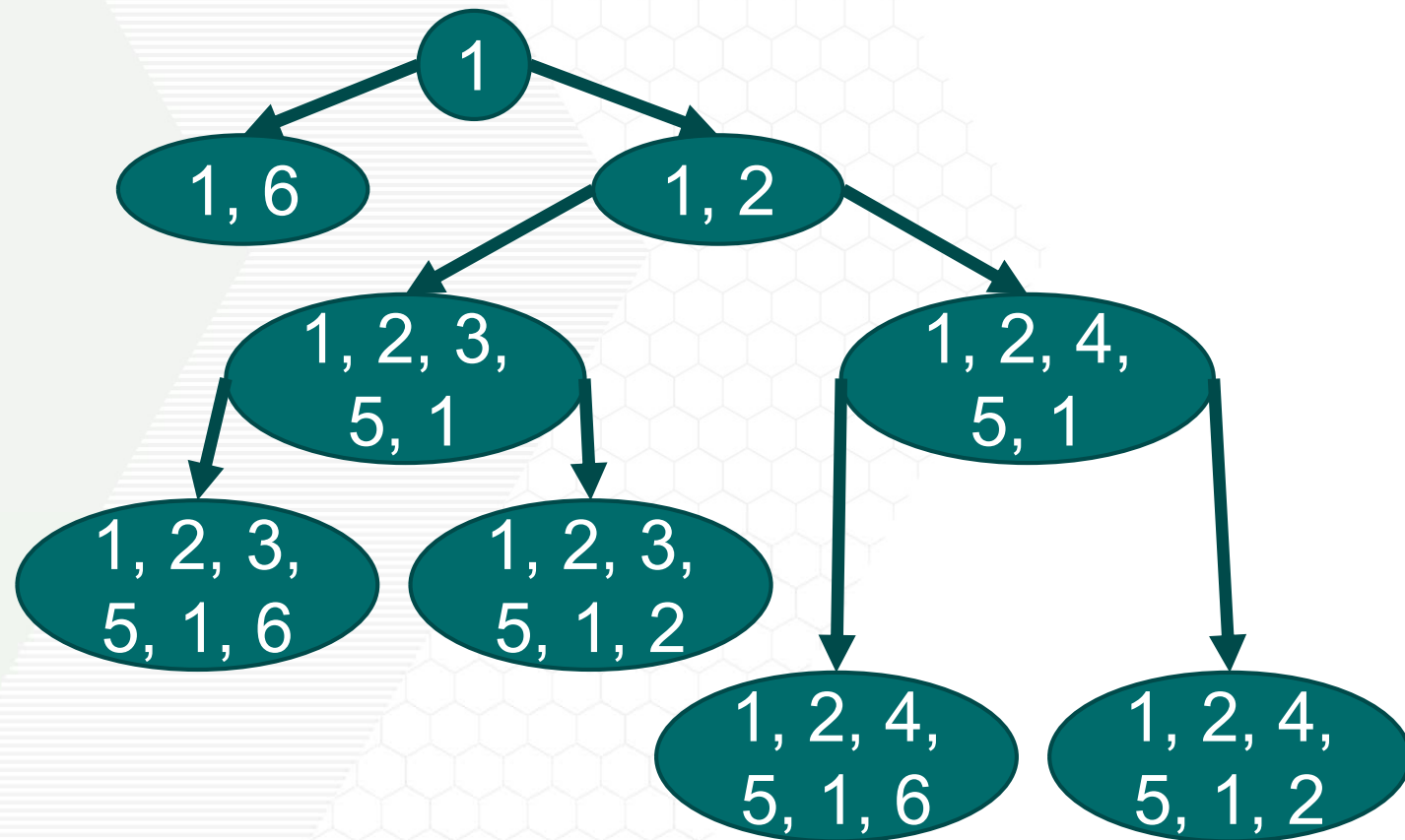
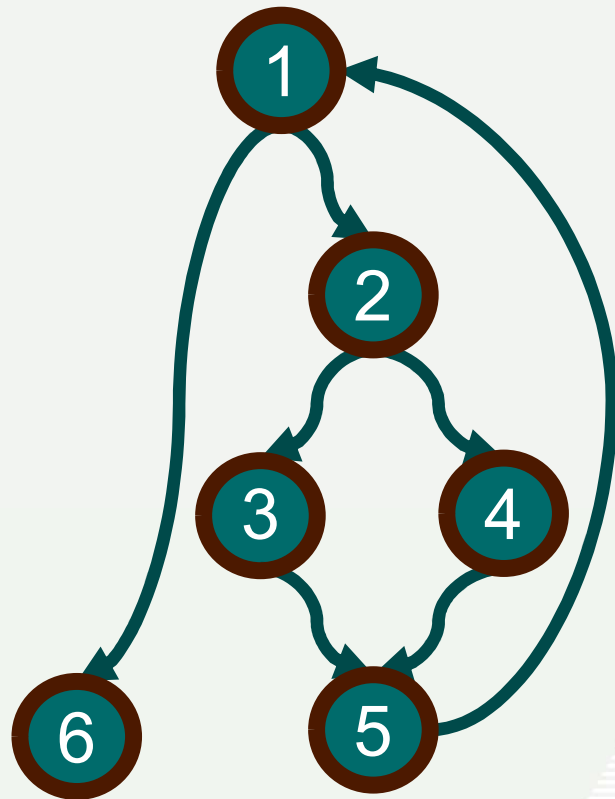
“Magical” Symbolic Analysis Engine

```
y = read()  
y = 2 * y  
if (y == 12)  
    fail()  
printf("OK")
```



- What input will cause the program to execute the fail function?
- The symbolic analysis engine will step through the program and create constraints on the values
- Constraints can then be queried to answer questions

- At each decision point the number of tracked states (possible program paths) doubles



EXAMPLE: VORTEX WARGAME

<http://www.overthewire.org/wargames/>



```
#include <...>
void print(unsigned char *buf, int len); // print state (for debugging)

#define e(); if(((unsigned int)ptr & 0xff000000)==0xca000000) { win(); }

int main() {
    unsigned char buf[512];
    unsigned char *ptr = buf + (sizeof(buf)/2);
    unsigned int x;

    while((x = getchar()) != EOF) {
        switch(x) {
            case '\n': print(buf, sizeof(buf)); continue; break;
            case '\\': ptr--; break;
            default: e(); if(ptr > buf + sizeof(buf)) continue; ptr++[0] = x;
        } } }
```

EXAMPLE: VORTEX WARGAME

<http://www.overthewire.org/wargames/>



```
#include <...>
void print(unsigned char *buf, int len); // print state (for debugging)

#define e(); if(((unsigned int)ptr & 0xff000000)==0xca000000) { win(); }

int main() {
    unsigned char buf[512];
    unsigned char *ptr = buf + (sizeof(buf)/2);
    unsigned int x;

    while((x = getchar()) != EOF) {
        switch(x) {
            case '\n': print(buf, sizeof(buf)); continue; break;
            case '\\': ptr--; break;
            default: e(); if(ptr > buf + sizeof(buf)) continue; ptr++[0] = x;
        } } }
```

EXAMPLE: VORTEX WARGAME

<http://www.overthewire.org/wargames/>



```
#include <...>
void print(unsigned char *buf, int len); // print state (for debugging)

#define e(); if(((unsigned int)ptr & 0xff000000)==0xca000000) { win(); }

int main() {
    unsigned char buf[512];
    unsigned char *ptr = buf + (sizeof(buf)/2);
    unsigned int x;

    while((x = getchar()) != EOF) {
        switch(x) {
            case '\n': print(buf, sizeof(buf)); continue; break;
            case '\\': ptr--; break;
            default: e(); if(ptr > buf + sizeof(buf)) continue; ptr++[0] = x;
        } } }
```


EXAMPLE: VORTEX WARGAME

<http://www.overthewire.org/wargames/>



```
#include <...>
void print(unsigned char *buf, int len); // print state (for debugging)

#define e(); if(((unsigned int)ptr & 0xff000000)==0xca000000) { win(); }

int main() {
    unsigned char buf[512];
    unsigned char *ptr = buf + (sizeof(buf)/2);
    unsigned int x;

    while((x = getchar()) != EOF) {
        switch(x) {
            case '\n': print(buf, sizeof(buf)); continue; break;
            case '\\': ptr--; break;
            default: e(); if(ptr > buf + sizeof(buf)) continue; ptr++[0] = x;
        } } }
```

EXAMPLE: VORTEX WARGAME

<http://www.overthewire.org/wargames/>



```
#include <...>
void print(unsigned char *buf, int len); // print state (for debugging)

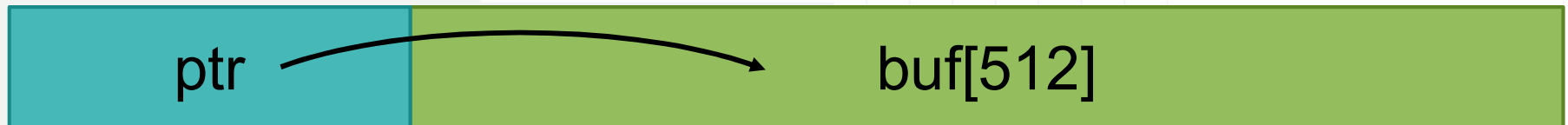
#define e(); if(((unsigned int)ptr & 0xff000000)==0xca000000) { win(); }

int main() {
    unsigned char buf[512];
    unsigned char *ptr = buf + (sizeof(buf)/2);
    unsigned int x;

    while((x = getchar()) != EOF) {
        switch(x) {
            case '\n': print(buf, sizeof(buf)); continue; break;
            case '\\': ptr--; break;
            default: e(); if(ptr > buf + sizeof(buf)) continue; ptr++[0] = x;
        } } }
```

EXAMPLE: VORTEX WARGAME

<http://www.overthewire.org/wargames/>



```
switch (input) {           // read char from user
case '\n': debug()         // print debug information
case '\\': ptr--;          // decrement ptr
default:
    if (ptr & 0xff000000 == 0xca000000) win();
    if (ptr < buf[len]) ptr++[0] = input;
}
```

- 3 decision points for each character in the input!
- Problem size: 3^n states being tracked for input size n !!

REASONS FOR STATE EXPLOSION



- Too much input/output data
 - Can we limit symbolic analysis to only the data we care about?
 - <10 symbolic bytes
 - E.g., an address, offset, or pointer
 - 20-80 symbolic bytes
 - E.g., shellcode, ROP chain
 - >200 symbolic bytes
 - E.g., shellcode plus data, long ROP chains, or complete data structures
- Too much included state
 - Can we limit symbolic state that gets tracked?
 - How to choose what state to prune?
- Too much executed code
 - Tracking every operation inflates the constraints, can we divide and conquer?

CONCOLIC EXECUTION TO THE RESCUE!



- Combine concrete and symbolic execution:
- **Concrete** + Symbolic = **Concolic**
- Use concrete execution with a concrete input to guide symbolic execution
- Selectively replace symbolic variables with concrete values
- Concrete values help simplify complex and unmanageable symbolic expressions

CONCOLIC EXECUTION EXAMPLE

- What value of z will cause the program to execute the fail function?

“Magical” Concolic Analysis

State
Split??

Concrete State

Symbolic

Symbolic State

```
y = read()  
z = read()  
y = 2 * y  
if (y >= 12)  
    if (z <= 20)  
        fail()  
printf("OK")
```

```
y = 8  
z = 42  
y = 16  
16 > 12
```

```
i0 <- "y"  
i1 <- "z"  
i0 <- 2 * i0  
i0 >= 12
```

```
i0 <- "y"  
i1 <- "z"  
i0 <- 2 * i0  
i0 < 12
```

CONCOLIC EXECUTION EXAMPLE

- What value of z will cause the program to execute the fail function?

No!
Concretize i_0

```
y = read()  
z = read()  
y = 2 * y  
if (y >= 12)  
    if (z <= 20)  
        fail()  
printf("OK")
```

```
y = 8  
z = 42  
y = 16  
16 > 12  
42 > 20  
printf("OK")
```

"Magical" Concolic Analysis

Concrete State

Symbolic

State Split!

```
i0 <- "y"  
i1 <- "z"  
i0 <- 16  
16 > 12  
i1 <= 20  
fail()  
  
i1 <= 20
```

```
i0 <- "y"  
i1 <- "z"  
i0 <- 16  
16 > 12  
i1 > 20  
printf("OK")  
  
i1 > 20
```

REWIND: CAREFULLY CONSIDER CODE COVERAGE!

- What value of z will cause the program to execute the fail function?

"Magical" Concolic Analysis

State
Split??

Concrete State

Symbolic

Symbolic State

```
y = read()  
z = read()  
y = 2 * y  
if (y >= 12)  
    if (z <= 20)  
        fail()  
printf("OK")
```

```
y = 2  
z = 42  
y = 4  
4 < 12
```

```
i0 <- "y"  
i1 <- "z"  
i0 <- 2 * i0  
i0 >= 12
```

```
i0 <- "y"  
i1 <- "z"  
i0 <- 2 * i0  
i0 < 12
```

REWIND: CAREFULLY CONSIDER CODE COVERAGE!

- What value of *z* will cause the program to execute the fail function?

No!
Concretize i0

“Magical” Concolic Analysis Engine

Concrete State

Symbolic State

```
y = read()
z = read()
y = 2 * y
if (y >= 12)
  if (z <= 20)
    fail()
printf("OK")
```

```
y = 2
z = 42
y = 4
4 < 12

printf("OK")
```

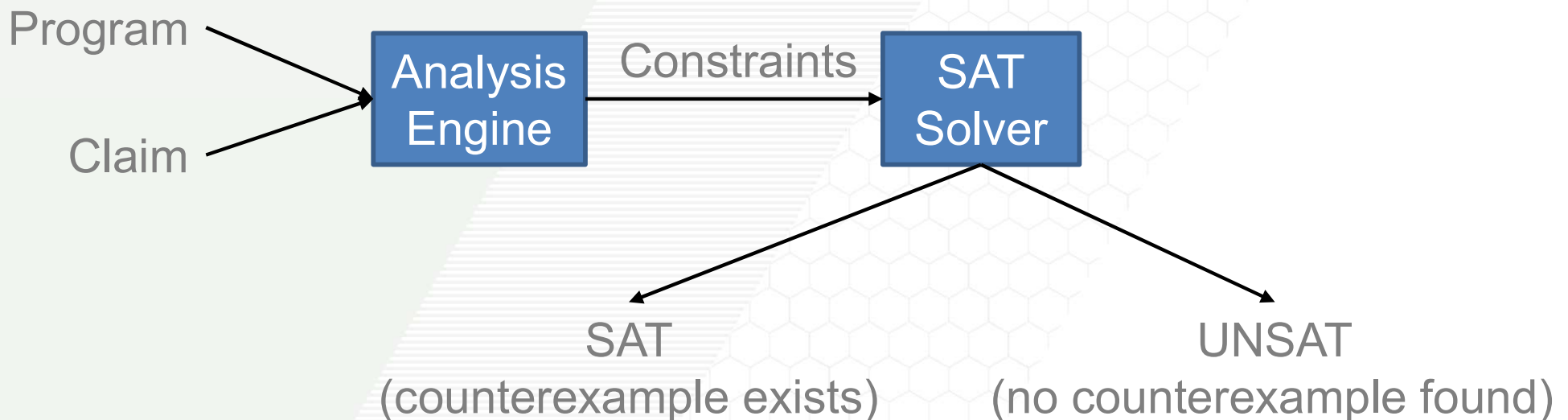
```
i0 <- "y"
i1 <- "z"
i0 <- 4
4 < 12

printf("OK")

i1 = ????
```

- Angr
 - Solving CTF problems, exploit generation, etc.
 - We use Angr in my lab often
 - <http://angr.io/>
- S2E: Selective Symbolic Execution
 - Automatic testing of binary code
 - <http://dslab.epfl.ch/proj/s2e>
- Triton
 - Leverages execution traces from many platforms
 - <https://triton.quarkslab.com>
- KLEE
 - Bug finding in source code
 - KLEE was very widely used until Angr
 - <http://ccadar.github.io/klee/>
- FuzzBALL
 - PoC exploits for given vulnerability conditions
 - <http://bitblaze.cs.berkeley.edu/fuzzball.html>
- Many more!
 - <https://github.com/ksluckow/awesome-symbolic-execution>

- In computational complexity theory, satisfiability (SAT) is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE
- **Main Idea:** Given a program and a claim, use a SAT Solver to find whether there exists an execution that violates the claim (a counterexample)
 - First problem shown to be NP-complete (1971)



- For each path:
 1. Convert the set of constraints into a Boolean formula (by ANDing all the variable constraints together)
 2. Check path condition satisfiability (SAT Problem), explore only feasible paths
 3. When execution path diverges, fork, adding constraints on symbolic values
 4. When we terminate (or crash), use a constraint solver to generate concrete input
- All symbolic analysis engines put time-limits on constraint solving
 - Often 10 hours...

Program

```
int x;  
int y=8, z=0, w=0;  
if (x)  
    z = y - 1;  
else  
    w = y + 1;  
assert (z == 7 ||  
        w == 9)
```

Constraints

```
y = 8 &&  
z = x ? y - 1 : 0 &&  
w = x ? 0 : y + 1 &&  
z != 7 &&  
w != 9
```


SAT

Can the assert statement fail?

no counterexample
assertion always holds!

Program

```
int x;  
int y=8, z=0, w=0;  
if (x)  
    z = y - 1;  
else  
    w = y + 1;  
assert (z == 5 ||  
        w == 9)
```



Constraints

```
y = 8 &&  
z = x ? y - 1 : 0 &&  
w = x ? 0 : y + 1 &&  
z != 5 &&  
w != 9
```

UNSAT

Can the assert statement fail?

counterexample:

```
y = 8, x = 1, w = 0, z = 7
```

AN EXAMPLE IN Z3



- SAT Solving has become very common in many applications
- Z3 is a high-performance theorem prover developed at Microsoft Research
- Z3 supports real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers

```
#!/usr/bin/env python
# Copyright (c) Microsoft 20
from z3 import *
```

```
x = Real('x')
y = Real('y')
s = Solver()
s.add(x + y > 5, x > 1, y > 1)
print(s.check())
print(s.model())
```

```
m = s.model()
for d in m.decls():
    print "%s = %s" % (d.name(), m[d])
```

```
~/z3/examples/python$ ./example.py
sat
[y = 4, x = 2]
y = 4
x = 2
```


MINI SYMBOLIC EXECUTION ENGINE IN Z3

- Z3 can find the input that would trigger a crash
- Fork at every predicate and explore each side

```
~/z3/examples/python$ ./example.py
```

```
Fork - Line 6
```

```
[3216] assume (2*x == y)
[3217] assume ¬(2*x == y)
[3217] exit
```

```
Fork - Line 7
```

```
[3216] assume (y == x + 10)
[3218] assume ¬(y == x + 10)
[3218] exit
```

```
[3216] Traceback (most recent call last):
  File "./test_me.py", line 11, in <module>
    test_me(x, y)
  File "./test_me.py", line 7, in test_me
    assert False
```

```
AssertionError: x = 10, y = 20
```

```
[3216] exit
```

```
1.  #!/usr/bin/env python
2.  from mc import *
3.
4.  def test_me(x, y):
5.      z = 2 * x
6.      if z == y:
7.          if y == x + 10:
8.              assert False
9.
10. x = BitVec("x", 32)
11. y = BitVec("y", 32)
12. test_me(x, y)
```

<http://kqueue.org/blog/2015/05/26/mini-mc/>

CREATING THE NEXT®

QUESTIONS?

CREATING THE NEXT®