Practical Satisfiability Modulo Theories (SMT) Solving

Joshua Wang

What is SMT?

- “The **satisfiability modulo theories** (**SMT**) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality” - wikipedia

First Order Logic

- Expressive formal language system that breaks statements down into
  - Things
    - Constants: `x, y`
    - Functions: `foo(x), bar(y)`
  - Relationships
    - Predicates: `assert(x > y)`
  - Connectives
    - &&, ||, !
  - Quantifiers
    - ∀, ∃

SMT Problem

- Decision problem (question that returns T/F) for formulas expressed using first order logic that return True or False based on theories of arithmetic, lists, bit-vectors, arrays, etc.

What is an SMT solver?

- Constraint solver
  - Constraint = statement that specifies properties of a solution to be found
- Examples:
  - Z3
  - STP
  - Yices
  - Alt-Ergo

How do SMT solvers work?

- User specifies formula that must be satisfied
- Solver attempts to find solution that satisfies formula
- If solver can find solution, formula is said to be **satisfiable**
- If solver cannot find solution, formula is said to be **unsatisfiable**

Symbolic Execution

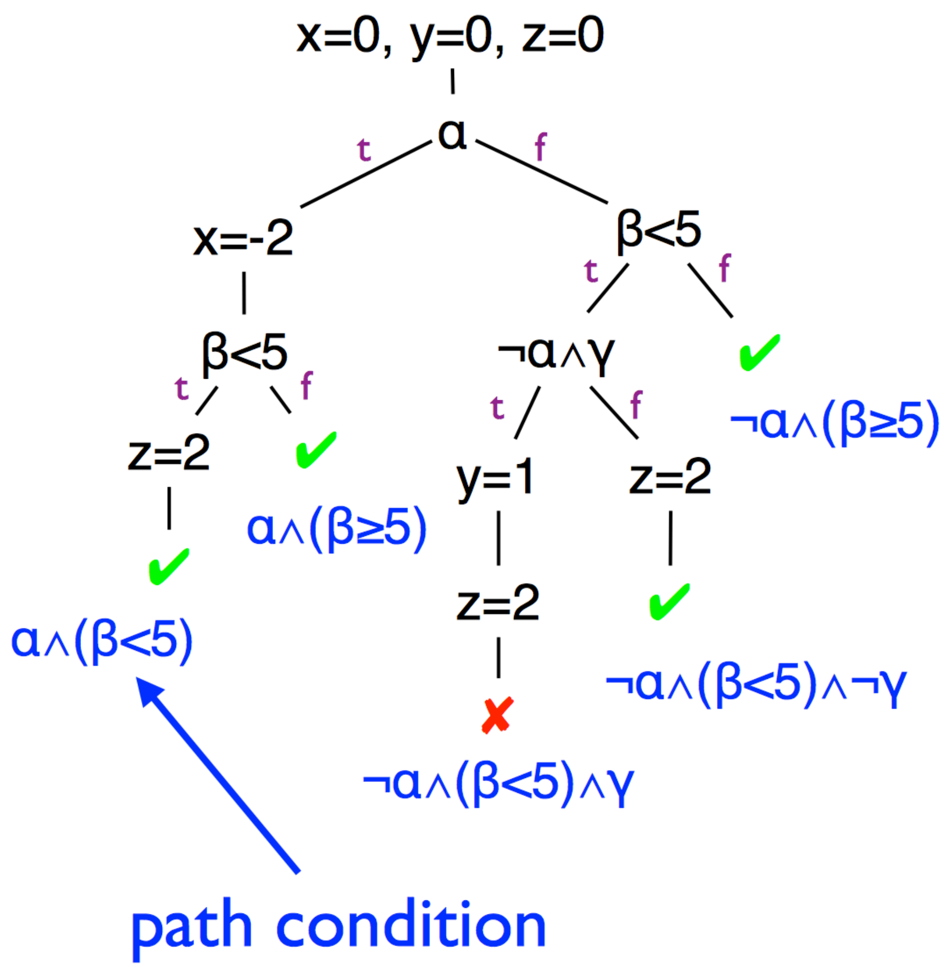
- Program analysis technique that treats input data as symbolic variables
- Creates expressions/ constraints based on symbolic variables
- Used in conjunction with constraint solver to generate new inputs/test cases (**concolic execution**)

```
int x = read();
x = x*2;
if (x == 10){
   foo();
}
```

```
s*2 == 10
```

Symbolic Execution Example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



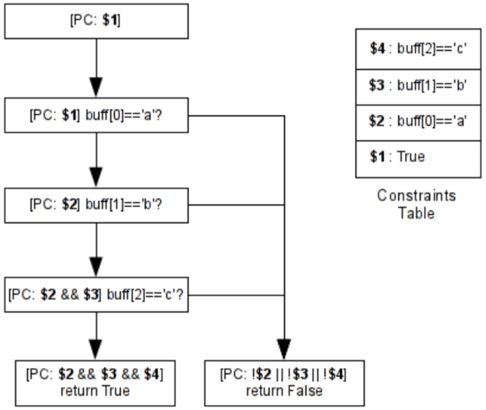
Source: http://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec13-SymExec.pdf

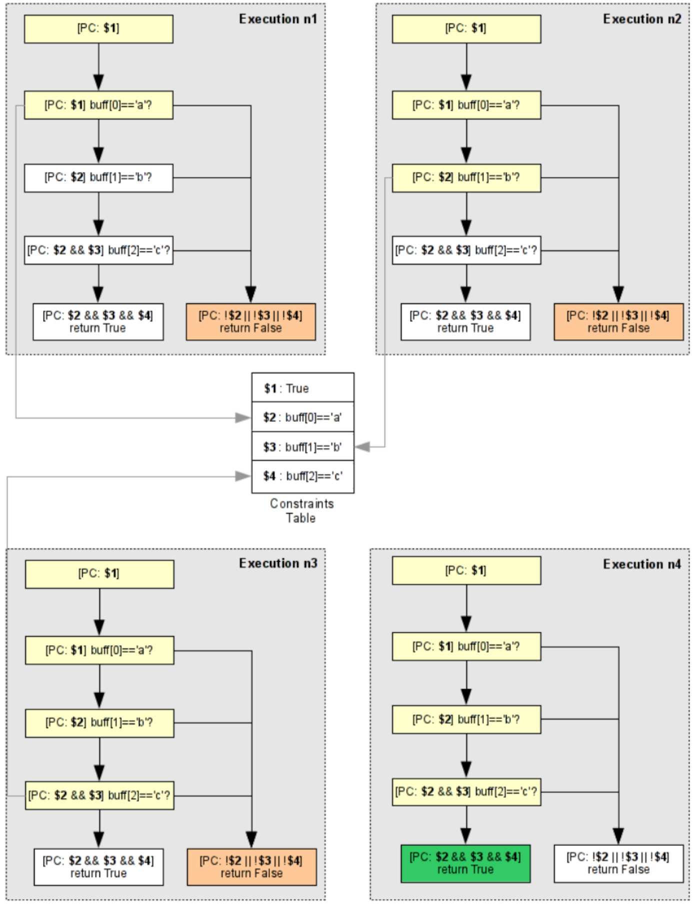
SMT Solvers and Software Security

- Uses constraints generated from symbolic execution
- Symbolic execution + constraint solving = **concolic execution**
- Used to aid in fuzzing, software verification
- Raison d'être = maximize code coverage

Concolic Execution

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>

int main(void)
{
  int   fd;
  char  buff[260] = {0};

  fd = open("serial.txt", O_RDONLY);
  read(fd, buff, 256);
  close(fd);

  if (buff[0] != 'a') return False;
  if (buff[1] != 'b') return False;
  if (buff[2] != 'c') return False;

  printf("Good boy\n");

  return True;
}
```



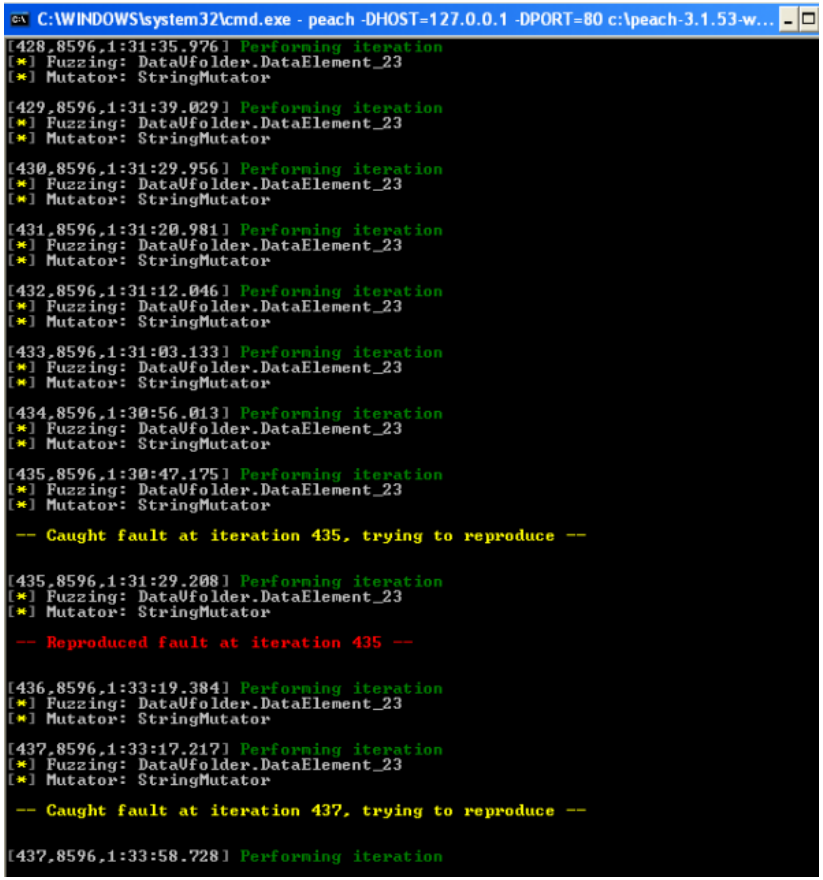


PC number	Constraints	return value
1	buff[0] != 'a'	return False
2	buff[0] == 'a' && buff[1] != 'b'	return False
3	buff[0] == 'a' && buff[1] == 'b' && buff[2] != 'c'	return False
4	buff[0] == 'a' && buff[1] == 'b' && buff[2] == 'c'	return True

Source: http://shell-storm.org/blog/Binary-analysis-Concolic-execution-with-Pin-and-z3/

Fuzzing

- Black box fuzzing
  - Don't have access to source
  - Input randomly generated
- White box fuzzing
  - Do have access to source
  - Reason about structure of code
  - Test cases generated intelligently



Black Box Fuzzing Limitations

- What are our chances of following code path down bar() using "dumb" fuzzing technique?
- 1 in 2^{32}
- slow! does not scale well.

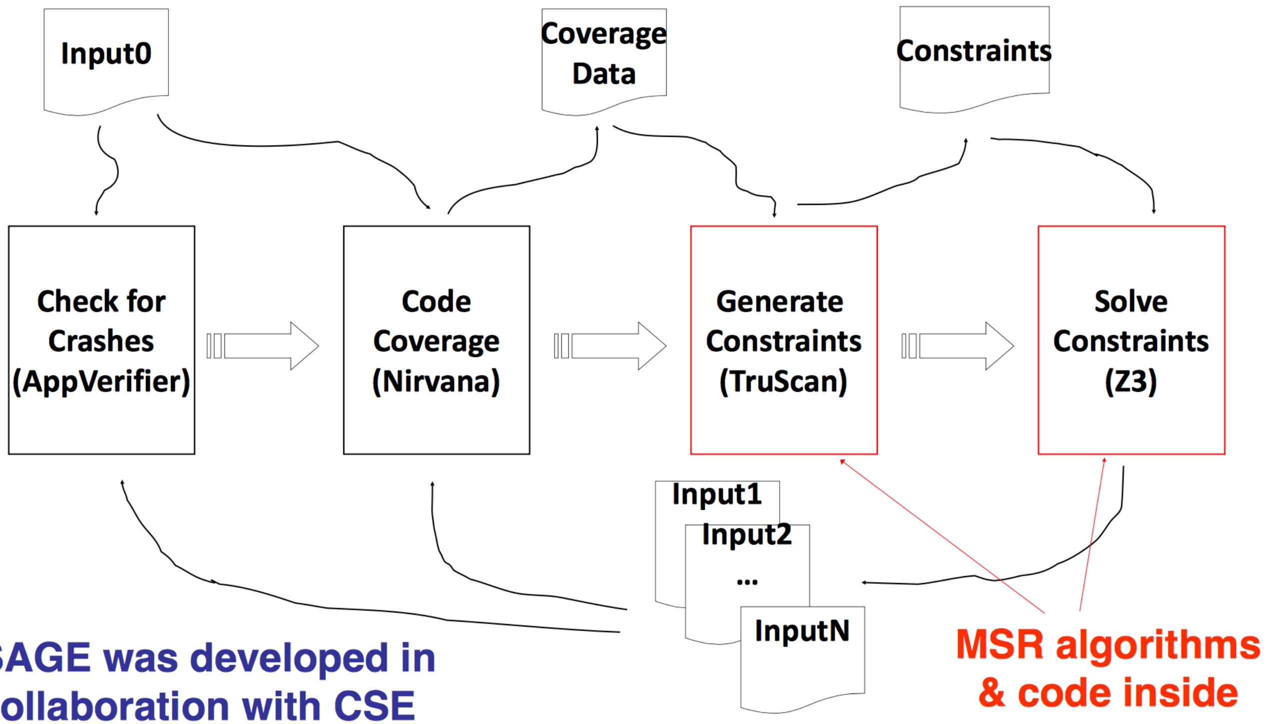
```
int foo(int x)
{
    int y = x+7;
    if( y == 24 ){
        bar();
    }
    return 0;
}
```

White Box Fuzzing

- Uses symbolic execution + constraint solving (**concolic execution**)
- Instead of using an actual value for input, treat it like a symbolic variable
- Extract constraints
- Solve linear equations to generate new input
- Use new input to follow new code path

SAGE

Basic idea:
1. Run the program with first inputs,
2. gather constraints on inputs at conditional statements,
3. use a constraint solver to generate new test inputs,
4. repeat - possibly forever!



SAGE was developed in collaboration with CSE

Source: http://research.microsoft.com/en-us/um/people/pg/public_psfiles/sage-in-one-slide.pdf

SAGE Impact

- Cleaned out 1/3 of Windows 7 bugs before release
- ANI animated cursor bug
- In theory it's supposed to find 100% of bugs, right?

Technique	Effort	Code coverage	Defects found
Combination of black box + dumb	10 min	50%	25%
Combination of white box + dumb	30 min	80%	50%
Combination of black box + smart	2 hr	80%	50%
Combination of white box + smart	2.5 hr	99%	100%

Source: https://msdn.microsoft.com/en-us/library/cc162782.aspx

Z3 (SMT-LIB Notation)

- `z3 -smt2 -in`
  - `reads commands from stdin`
- declare constants
  - `(declare-const x Int)`
- add constraints by adding assertions using **polish** notation
  - `(assert (> a 10))`
- check satisfiability
  - `(check-sat)`
- get interpretation that makes all formulae true
  - `(get-model)`

Unsatisfiable Example (SMT-LIB Notation)

$x > 0$

$y = x + 1$

$y < 0$

```
(echo "this should be unsatisfiable")
(declare-const x Int)
(declare-const y Int)
(assert (> x 0))
(assert (= y (+ x 1)))
(assert (< y 0))
(check-sat)
(get-model)
```

Unsatisfiable Example (Python Bindings)

```
x > 0
y = x + 1
y < 0
```

```python
from z3 import *

x = Int("x")
y = Int("y")

s = Solver()

s.add(x > 0)
s.add(y == x+1)
s.add(y < 0)

print s.check()
print s.model()
```

Solving Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Concolic Testing Conclusions

- Majority of RCE vulns found in past several years still attributed to dumb fuzzing techniques
- While Z3 is open source, code for projects like Mayhem and Sage and closed-source. Atypical in security community. (*Kerckhoff's principle*)
- Difficult to determine the efficiency of SAGE since it's closed source